Architectural considerations for a control system processor

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 of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/11075](https://dspace.lboro.ac.uk/2134/11075)

Publisher: © Dipesh I. Patel

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
ARCHITECTURAL CONSIDERATIONS FOR A
CONTROL SYSTEM PROCESSOR

by

Dipesh Ishwerbhai Patel

A Doctoral Thesis
Submitted in partial fulfillment of the requirements
for the award of the degree of

Doctor of Philosophy

of the Loughborough University of Technology

September 1996

© by Dipesh I. Patel 1996
DEDICATION

to the ones I love,

*Mum, Dad*

&

*Nirupa*
Modern design methodologies for control systems create controllers with dynamics which are of a similar order to the physical system being controlled. When these are implemented digitally as Infinite Impulse Response (IIR) filters the processing requirements are extensive, in particular when high sample rates are necessary to minimise the detrimental effects of sample delay.

The aim of the research was to apply signal processing techniques to facilitate the implementation of control algorithms in digital form, with the principal objective of maximising the computational efficiency, either to achieve the highest possible sample rates using a given processor, or to minimise the processor complexity for a given requirement. One of the approaches is to design a fixed point processor whose architecture is optimised to meet the computational requirements of signal processing for control, thereby maximising what can be achieved with a single processor.

Hence the aim of the research was to head towards a processor architecture optimised for Control System Processing. The design of this processor is based on a unified structural form and it will be shown that controllers, represented either in state space form or as transfer functions, can be implemented using this unified structure. The structure is based on the $\delta$-operator, which has been shown to be robust to changes in coefficients and hence require shorter coefficient wordlength to achieve a comparable performance to traditional $z$-operator based structures. Additionally, the $\delta$-operator structures are also shown to have lower wordlength requirements for the internal variables.
Also presented is a possible architecture for a Control System Processor and a model for the processor is developed and constructed using VHDL. This is simulated on a test bench, also designed in VHDL. The results of implementing a phase advance controller on the processor are then compared with those obtained from a MATLAB simulation.

Keywords:
Control Processors
Digital Controller Implementation
δ-operator
Structures for Implementation
δ-operator state space structures
ASIC's
VHDL
Architectures for Control Processors
ACKNOWLEDGEMENTS

The author wishes to thank his supervisor Professor Goodall without whose vision this research would not have been possible; I would like to express my gratitude to him as his guidance and help during the course of the research has been priceless.

A big thank you to the Committee of Vice Chancellors and Principals (CVCP) for having faith in me and awarding me the precious Overseas Research Studentship (ORS) Award.

I also wish to express my appreciation to the members of the Control Group, particularly John Pearson, Pete 'Shaggy' Holme, Jonathan Paddison and Ian Pratt, for the constructive discussions we had together; Mustafa 'Regy' Abuzeid, Malcolm Fraser and Mike Oliver for being their usual entertaining selves.

Thanks are also due to the residents of 61, Mountfields Drive (Keval, Pips, Ro) who made my two years there enjoyable, and especially Doug Green for the inspiring discussions we had regarding work, life, cricket, and all that is under the sky!!

During the course of my research a number of people have drifted in and out of my life and they have been as much a part of my life as those mentioned above. A special mention among these to the residents of E26 (91/92) and E24 (93/94). Of these, Anoop, Hari and Sam have left a very deep impression.

A big thankyou to: Martin Jansz for proof reading a very important chapter and the constructive suggestions, my brother Jignesh for helping me with the formatting at a time when I was struggling with the work load, my sister Shefali and cousin Jay for the financial help during the rough days!

And last but not least God for helping me through some testing times.

Architectural Considerations for a CSP
# TABLE OF CONTENTS

DEDICATION _______________________________________________ i
ABSTRACT ______________________________________________ ii
ACKNOWLEDGEMENTS ____________________________________ iv
TABLE OF CONTENTS____________________________________ v
NOMENCLATURE________________________________________ ix

Chapter 1 INTRODUCTION____________________________________ 1
1.1 Fixed vs Floating Point_________________________________ 5
1.2 Analogue vs Digital____________________________________ 6
1.3 Digital Filters________________________________________ 7
1.4 Processors___________________________________________ 9
1.5 Summary____________________________________________ 11
1.6 Organisation of the Thesis_____________________________ 11

Chapter 2 OVERVIEW OF THE FIELD OF INTEREST______________ 14
2.1 Modelling of Dynamic Systems____________________________ 14
  2.1.1 Transfer function models_____________________________ 15
  2.1.2 State Variable Models_______________________________ 15
  2.1.3 Performance Specifications____________________________ 16
2.2 Design Methods________________________________________ 17
  2.2.1 Classical Control___________________________________ 17
  2.2.2 Optimal Control____________________________________ 18
  2.2.3 Robust Control (H etc.)______________________________ 18
  2.2.4 Unified Theory Approach____________________________ 18
  2.2.5 Other Methods____________________________________ 19
2.3 Types of Controllers____________________________________ 20
2.4 Implementation________________________________________ 21
  2.4.1 Discretization______________________________________ 21
  2.4.2 Structures_________________________________________ 22
    2.4.2.1 Wordlength Requirements___________________________ 23
    2.4.2.2 z-operator______________________________________ 25
    2.4.2.3 δ-operator______________________________________ 27
    2.4.2.4 Optimum Structures______________________________ 29
  2.4.3 Hardware Alternatives________________________________ 31
2.5 ASIC's and CSP's______________________________________ 35

Architectural Considerations for a CSP________________________ v
## Table of Contents

2.6 Summary.................................................................................................................. 38

Chapter 3  STRUCTURES FOR IMPLEMENTATION ..................................................... 40

3.1 The Internal Variables .............................................................................................. 41

3.2 The Cascaded Modified Canonic Structure ............................................................ 42
  3.2.1 Internal Variable Overflow .................................................................................. 43
    3.2.1.1 $l_\infty$-norm analysis ................................................................................... 44
    3.2.1.2 Example using $l_\infty$-norm ....................................................................... 45
    3.2.1.3 Scaling ......................................................................................................... 46
  3.2.2 Internal Variable Underflow ................................................................................ 47
    3.2.2.1 Round-off Noise Analysis .......................................................................... 48
    3.2.2.2 Steady State Analysis .................................................................................. 54
    3.2.2.3 Comparison of Analyses ............................................................................. 54

3.3 The Cross Coupled Modified Canonic Structure .................................................... 61
  3.3.1 Calculating the coefficient values ........................................................................ 63
    3.3.1.1 Complex roots ............................................................................................ 65
    3.3.1.2 Repeated roots (one pair only) ................................................................... 67
  3.3.2 Internal Variable Overflow .................................................................................. 68
    3.3.2.1 Initial and Final value analysis ................................................................... 69
    3.3.2.2 $l_\infty$-norm analysis .................................................................................. 73
  3.3.3 Internal Variable Underflow ................................................................................ 74
    3.3.3.1 Round-off Noise Analysis .......................................................................... 74
    3.3.3.2 Steady State Analysis .................................................................................. 76
    3.3.3.3 Comparison of Analyses ............................................................................. 81

3.4 Comparison of the Structures .................................................................................... 87

3.5 Summary .................................................................................................................... 91

Chapter 4  THE MODIFIED CANONIC STATE SPACE STRUCTURE ............................ 93

4.1 State Space Algorithms ............................................................................................ 94

4.2 The state space modified canonic structure ............................................................ 94

4.3 Eigenvalues ............................................................................................................... 96

4.4 Transformations to controller canonical form .......................................................... 98

4.5 The algorithm .......................................................................................................... 100
  4.5.1 The procedure .................................................................................................... 100
  4.5.2 Testing ............................................................................................................... 101
  4.5.3 Beware .............................................................................................................. 101

4.6 Implementing state space controllers ....................................................................... 102
  4.6.1 Continuous time models .................................................................................... 102
  4.6.2 Transforming to modified canonic form ......................................................... 103
  4.6.3 Example ............................................................................................................ 104
  4.6.4 Discrete models .................................................................................................. 106

4.7 Sensitivity Analysis .................................................................................................. 106
  4.7.1 Magnitude sensitivity ....................................................................................... 107
  4.7.2 $\delta$-operator models ....................................................................................... 108
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7.3 Choosing a frequency range _______________ 109</td>
</tr>
<tr>
<td>4.7.4 Example _______________ 109</td>
</tr>
<tr>
<td>4.8 Summary_____________________________ 112</td>
</tr>
<tr>
<td>Chapter 5 A CONTROL SYSTEM PROCESSOR (CSP) ____________ 113</td>
</tr>
<tr>
<td>5.1 The CSP in the Control Field _____________________ 114</td>
</tr>
<tr>
<td>5.2 Coefficients and Variables_______________________ 116</td>
</tr>
<tr>
<td>5.2.1 Coefficient Format ____________________________ 117</td>
</tr>
<tr>
<td>5.2.2 Variable Format______________________________ 120</td>
</tr>
<tr>
<td>5.3 The Instruction Set____________________________ 122</td>
</tr>
<tr>
<td>5.3.1 Instructions ________________________________ 123</td>
</tr>
<tr>
<td>5.3.2 Encoding Format______________________________ 124</td>
</tr>
<tr>
<td>5.4 The Core of the Processor_______________________ 126</td>
</tr>
<tr>
<td>5.4.1 The Registers ________________________________ 126</td>
</tr>
<tr>
<td>5.4.2 The Adder/Subtractor Unit (ASU)________________ 127</td>
</tr>
<tr>
<td>5.4.3 The Multiplier ______________________________ 128</td>
</tr>
<tr>
<td>5.4.4 A Generic Architecture_______________________ 132</td>
</tr>
<tr>
<td>5.5 Summary_____________________________ 135</td>
</tr>
<tr>
<td>Chapter 6 A MODEL FOR THE CSP ____________________ 136</td>
</tr>
<tr>
<td>6.1 The Bus Architecture___________________________ 137</td>
</tr>
<tr>
<td>6.1.1 The CSP Port ________________________________ 137</td>
</tr>
<tr>
<td>6.1.2 Memory Read ________________________________ 138</td>
</tr>
<tr>
<td>6.1.3 Memory Write_______________________________ 139</td>
</tr>
<tr>
<td>6.1.4 ADC Read _________________________________ 140</td>
</tr>
<tr>
<td>6.1.5 DAC Write______________________________ 141</td>
</tr>
<tr>
<td>6.2 The CSP Model______________________________ 142</td>
</tr>
<tr>
<td>6.2.1 Data Bus Connections________________________ 143</td>
</tr>
<tr>
<td>6.2.2 Coefficients_______________________________ 144</td>
</tr>
<tr>
<td>6.2.3 Interrupts_______________________________ 144</td>
</tr>
<tr>
<td>6.2.4 Initial State______________________________ 144</td>
</tr>
<tr>
<td>6.3 The Test Bench______________________________ 145</td>
</tr>
<tr>
<td>6.3.1 The Clock Generator________________________ 145</td>
</tr>
<tr>
<td>6.3.2 The ADC Model____________________________ 146</td>
</tr>
<tr>
<td>6.3.3 The DAC Model____________________________ 146</td>
</tr>
<tr>
<td>6.3.4 The RAM Model____________________________ 146</td>
</tr>
<tr>
<td>6.3.5 The EPROM Model__________________________ 147</td>
</tr>
<tr>
<td>6.4 Implementing a Phase Advance Controller__________ 147</td>
</tr>
<tr>
<td>6.4.1 The Coefficients___________________________ 148</td>
</tr>
<tr>
<td>6.4.2 The Variables____________________________ 149</td>
</tr>
<tr>
<td>6.4.3 The Equations____________________________ 149</td>
</tr>
<tr>
<td>6.4.4 Software______________________________ 150</td>
</tr>
<tr>
<td>6.4.5 Comparison with MATLAB_____________________ 151</td>
</tr>
<tr>
<td>6.5 Synthesis_____________________________ 155</td>
</tr>
<tr>
<td>Section</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6.6 Summary</td>
</tr>
<tr>
<td>Chapter 7  CONCLUSIONS &amp; SUGGESTIONS</td>
</tr>
<tr>
<td>7.1 Contributions of Thesis</td>
</tr>
<tr>
<td>7.2 Conclusions</td>
</tr>
<tr>
<td>7.3 Suggestions for Further Work</td>
</tr>
<tr>
<td>REFERENCES</td>
</tr>
<tr>
<td>APPENDICES</td>
</tr>
</tbody>
</table>
**NOMENCLATURE**

- $A_{\delta n}, B_{\delta n}, C_{\delta n}, D_{\delta n}$: State Space Matrices derived in $\delta$-operator form, represented in modified canonic form
- $A_z, B_z, C_z, D_z$: State Space Matrices derived in $z$-operator form
- ADC: Analogue to Digital Converter
- ASIC: Application Specific Integrated Circuit
- ASU: Adder/Subtractor Unit
- $a_i$: Denominator Coefficients of transfer functions $G(s)$ and $G(z)$
- $b_i$: Numerator Coefficients of transfer functions $G(s)$ and $G(z)$
- $b$: Number of Fractional bits for underflow analysis
- CLA: Carry Look Ahead Adder
- CSP: Control System Processor
- $\delta$: Delta Operator ($= z^{-1}$)
- DAC: Digital to Analogue Converter
- DSP: Digital Signal Processor
- $e(nT)$: Error sequence used for round-off noise analysis
- EPROM: Erasable Programmable Read Only Memory
- FPGA: Field Programmable Gate Array
- $f_s$: Sampling Frequency
- $G(s)$: Transfer function in $s$-domain
- $G(\delta)$: $\delta$-operator transfer function
- $G(z)$: $z$-operator transfer function
- $H^2$: r.m.s transfer function magnitude between noise source and output
- $h(nT)$: Impulse Response Sequence between the internal variable and input $u$
- $I$: Identity Matrix

*Architectural Considerations for a CSP*
### Nomenclature

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTI</td>
<td>Linear Time Invariant</td>
</tr>
<tr>
<td>MISO</td>
<td>Multiple Input Single Output</td>
</tr>
<tr>
<td>(p, q, r, d_{1}, d_{2})</td>
<td>Coefficients of a second order modified canonic structure</td>
</tr>
<tr>
<td>(p_{1}, p_{2}, q_{1}, q_{2}, d_{11}, d_{12}, d_{21}, d_{22})</td>
<td>Coefficients of the cross coupled modified canonic structure</td>
</tr>
<tr>
<td>PABX</td>
<td>Personal Automated Branch Exchange</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional Integral plus Derivative</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulated</td>
</tr>
<tr>
<td>(r.m.s)</td>
<td>root mean square</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>SISO</td>
<td>Single Input Single Output</td>
</tr>
<tr>
<td>(s)</td>
<td>Laplace Operator</td>
</tr>
<tr>
<td>(S_{\delta})</td>
<td>Sensitivity Matrices for (\delta)-operator form</td>
</tr>
<tr>
<td>(S_{z})</td>
<td>Sensitivity Matrices for (z)-operator form</td>
</tr>
<tr>
<td>(T)</td>
<td>Transformation Matrix for state space systems</td>
</tr>
<tr>
<td>(T_{s})</td>
<td>Sampling Interval</td>
</tr>
<tr>
<td>(v, w, x)</td>
<td>Internal Variables of the second order modified canonic structure</td>
</tr>
<tr>
<td>(v_{1}, v_{2}, w_{1}, w_{2})</td>
<td>Internal Variables of the cross coupled modified canonic structure</td>
</tr>
<tr>
<td>(u)</td>
<td>Input to the controller</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
<tr>
<td>(y)</td>
<td>Output from the controller</td>
</tr>
<tr>
<td>ZOH</td>
<td>Zero Order Hold</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

The invention of the transistor in 1947 sparked off an unprecedented revolution in electronics. It brought the digital age into the foreground and in a relatively short time technology evolved so greatly that fabrication of an entire processor on a piece of silicon, no bigger than a thumb nail, was now possible and the microprocessor was born in the early 1970's. Since then microprocessors have captured the imagination of many fields and microprocessor based systems design has become a discipline in its own right.

One of the first applications of the microprocessor was the digital calculator but the most popular application, of course, of the microprocessor is the personal computer (PC), and many homes in the western world have a PC which is used for a variety of tasks. However, the first ever programmable computing machine was designed in 1834, more than a century before any electronic computer. It was the Analytical Engine of Charles Babbage. Only a tiny fragment of the machine was ever built, but Babbage's drawings and notes show that the design was very much along the lines of a modern computer, with a central processor, a memory unit, and a punched card reader for the program. It also had a printer for the output.

Back in the 1970s, the first 8-bit general purpose microprocessors appeared on the scene such as the Intel 8080. These were slow and cumbersome by today's standards but showed where the future lay. They were generally too expensive, large, and slow to be of use in specific real-time control systems - they used custom designed electronic or mechanical controllers - and hence found applications in offices and laboratories as computers dedicated to specific computational tasks. For example in large test and measurement systems, computers were used to monitor
and record but not control the systems. After the recording was over, the computer would be taken offline and used to analyse the data. One of the earliest applications of computers actually to control and react to a system was traffic light control. In large towns and cities, a central computer was used to control the traffic lights and the software was specifically written using knowledge of the town and likely traffic patterns [Godfrey (1996)].

Around this time, manufacturers started making use of general purpose 8-bit microprocessors in what are now called embedded systems. Dedicated software was written for these microprocessors to run on a purpose built platform. The application category included printers, early centralised telecommunication systems and small PABX systems. These early microprocessors required additional memory and other peripheral chips and meant that the equipment was fairly large but fast [Godfrey (1996)].

Over the years processor design methods and fabrication technology have evolved to such an extent that it is now possible to fabricate processors that are capable of carrying out tasks that were thought unimaginable only a few years ago. In fact, the microprocessors used in these embedded systems today normally have the additional memory and other peripheral devices included on the one chip, thereby minimising size and power consumption: that is they are now integrated microprocessors. Most embedded systems still use 8- or 16-bit microprocessors and it is only for the top end where one finds 32-bit microprocessors being used. These are used in equipment such as printers, scanners and control of high performance communication equipment.

Another class of microprocessors - as opposed to general purpose microprocessors described so far - is the Digital Signal Processor (DSP). DSPs are increasingly being used in many applications due to their number processing capability and it is increasingly becoming evident that these are also going to be produced as integrated devices: that is with additional memory and peripherals.
The future looks even more varied as the limit of what can be achieved on a silicon chip, in terms of speed and power, is rapidly approached [Wilkes (1996)]. Scientists are reporting developments using plastics, DNA and even bacteria to process information. Some of the ideas are still tentative research projects but some are taking their first steps into commercial exploitation. The options for speeding up the processing are: to either pack the transistor's closer together, the limit for this is also being approached rather rapidly, or switch to a faster material such as gallium-arsenide (Ga-As). Ga-As devices are significantly faster, have low power requirements and are already used in high-frequency communication circuits. However, as a large volume computer chip material, it has proved too difficult and expensive to be a viable contender. A more promising alternative seems to be silicon-germanium technology. Even though the ideas for this technology dates back to the 1950s, its only recently that engineers have actually found ways of making these devices.

Amongst other more exotic ideas [Weber (1995)], there are:

- **the Fantastic Plastic**, a iodine doped polyacetylene which has not only a metal like appearance and but also conducts electricity like one. Semiconducting plastics have been used to make batteries, light emitting diodes, solar cells and even plastic transistors but they have proved difficult to produce and can be unstable. Plastic microprocessors are not yet available.
- **Optical Computing**. This has been much talked about recently especially with respect to parallel computing. The central idea is that photons do not interact with each other as electrons do and hence it is easier to process a million bitstreams simultaneously. Since the photons do not interact, beams of light can pass through each other and emerge unchanged. Additionally, an optical gate can switch one beam just as easily as it can a number of beams at the same time; the only limit being the system's optical resolution. It is expected that optical computing will be a commercial reality soon and there is talk of a 20GHz palm-sized version, even though at present the maximum achievable speed is around 50MHz - around the speed of an average PC.
• **Bio-computing.** There is growing confidence in this area as illustrated by the development of three dimensional optical memories using a protein called bacteriorhodopsin which is a light sensitive chemical and can be repeatedly switched between two different molecular forms when it absorbs either green or blue light. In principle bacteriorhodopsin memories could store nearly 100GB in one cubic centimetre and access it in a few picoseconds!

• Finally, there is the prospect of using DNA to solve computational problems. It has been estimated that molecular computers conceivably could execute more than a thousand trillion operations per second whereas the current supercomputers can execute about a trillion operations per second! Additionally, the molecular components might be as much as a billion times more energy efficient than current electronic computers.

It makes one wonder to what extent the above concepts can be extended and whether true machine intelligence, that is a machine that thinks for itself and hence designs, builds, modifies and repairs itself, is far off! If so, are the scenarios depicted in popular films like Terminator celluloid fantasy or a feasible future reality? But that is all for the future, at present there is the microprocessor for which the processing power is rapidly increasing and ...

One of the areas which the microprocessor has affected greatly is the control field. Control has become a well recognised branch of engineering over the past thirty years establishing itself over a vast range of disciplines and applications. The advent of the microprocessor has increased the range further and this is expanding rapidly such that embedded systems profoundly affect one’s everyday life, whether they are in the cars one drives or the aeroplanes on which one flies to business and vacation destinations, or a myriad of other products such as intelligent gas pumps, interactive television and dishwashers. The power of these microprocessors is also increasing at a phenomenal rate but the microprocessors providing the capability for this intelligence appear as two main types: floating point and fixed point.
II Fixed vs Floating Point

Ideally, every application would use a floating point processor because its numeric format makes it ideal for performing complex mathematical operations where performance is the primary concern. However, competitive cost pressures and the price sensitivity of high volume products dictate the use of fixed point processors for many embedded applications.

Compared to floating point processors, fixed point processors generally contribute to lower overall system costs, reduced power consumption, and a smaller package size. However, the higher software design and associated development costs associated with fixed point processors partially offset these advantages [Larimer and Chen (1995)].

The finite nature of such fixed point implementations has meant that the digital techniques have not proved easy to apply, and a greater number of decisions need to be made when designing a digital controller than when designing an analogue controller. Often engineers have to make several iterations before a required specification can be met.

The fundamental ideas of control, however, remain the same with digital components, and anyone with an intuition for control systems will find that good solutions can still be achieved. The task is however more difficult and the problems surface when one turns to mathematics to confirm intentions or fine tune solutions. Digital techniques have meant that the engineers had to use the unfamiliar z-domain algebra for the discrete systems and not the familiar s-domain algebra used for continuous systems. This has instilled a lack of confidence in some engineers when using the mathematics required for digital control.
Introduction

1.2. Analogue vs Digital

With all the problems associated with digital controllers, the question is: What is the motivation for using a digital controller rather than an analogue controller? A qualitative comparison between analogue and digital controllers has been made by [Forsythe and Goodall (1991)].

Factors such as choice of processors, tools that will be needed, component count with increasing controller complexity, complexity of printed circuit board (PCB) design are central issues behind each of the two controller choices. Other factors which have to be considered are sample delays, computational delays and quantisation effects that are introduced in the digital controller. The combination of these latter factors almost invariably means that the performance of the digital controller will ultimately be inferior to an equivalent analogue controller, especially if the latter option is feasible as a proposition in the circumstances. The end result is that the design of an effective digital controller is not a trivial process when compared to an equivalent analogue one. However, there are reasons for resorting to using a digital solution:

1/ Lower cost - As the controller complexity increases, the analogue solution results in an increasing component count while for the digital solution, a single digital controller can be programmed to carry out the tasks in sequence. This has a potential for lower cost particularly in lower-speed systems.

2/ Adaptive control - Where typically, the system is monitored and the control parameters and/or structure are adapted in order to maintain an optimal performance. While this is a theoretical possibility using analogue components, it is practically quite difficult to implement. If in a particular application an adequate or comparable performance can be achieved using a digital system, and if there is spare computational capacity, then the monitoring and implementation of the tasks necessary for the adaptive
control can be performed during the spare capacity. So the adaptability of digital controllers may override other considerations.

3/ Volume production - If a product involving a control system is required in large quantities, it might be commercially attractive to integrate the product onto full- or semi-custom silicon (an Application Specific Integrated Circuit (ASIC)). This is more straightforward using digital ASIC's, though one must remember the analogue interface requirements of digital controllers. Until recently digital ASIC's were the only possibility but it is now possible to implement analogue circuits on ASIC's (mixed-mode designs) and hence the analogue interface requirements can also be included along with the digital components. It is also possible to produce full custom analogue ASIC's as well, though the size of the circuit which can be placed on an analogue ASIC remains small compared to what can be achieved with digital ASIC's. The cost of producing such an analogue ASIC also remains high at present when compared to the production of a digital ASIC.

1.3 Digital Filters

Having discussed the issue of analogue versus digital controllers, the next stage to understand in implementing digital controllers is the digital filters issue. The term digital filters is borrowed from the communications (signal processing) industry. Essentially, classical control systems require compensation stages such as phase advance, lag-lead, proportional plus integral (PI), 'notch' filters, and a number of other more specialised transfer functions.

It is widely accepted that recursive digital filters operating at sample frequencies significantly in excess of their dominant frequencies are most appropriate for such requirements. A minimum factor of 10 [Hanselmann (1987)] is often recommended, although for high performance systems, in which speed of response to inputs may be at a premium, factors of 30 or higher may be needed in order to keep the phase
lag associated with sampling and computational delays to a minimum.

This high sampling rate (relative to the filter’s characteristic frequencies) creates structural problems within the digital filter sections, resulting in high coefficient sensitivity and long wordlengths for internal variables which must be recognised and allowed for in each filter design. Both of these features create a need for high precision arithmetical operations which at the very least need to be understood, but which may also place a heavy load on the computational capacity of the processor.

The real problem is the \textit{z-operator} that is traditionally used in the design of digital filters. It becomes increasingly inappropriate to use it as the sample rate increases but fortunately an alternative approach based on the \textit{d-operator} exists [Goodwin (1985), Goodall and Brown (1985)]. This totally avoids the coefficient sensitivity problems, and there are additional advantages related to different structures. One of these structures is such that it creates a uniformity in the scaling of the internal variables, hence leading to shorter wordlengths, culminating in the use of fixed point variables with low precision floating point coefficients.

Modern control systems designed with state space techniques and using full state feedback do not require the implementation of recursive filters when digitally implemented, although the gain matrix applied to the state variables can benefit computationally from relatively low precision coefficients and longer variables. However, access to all the state variables is rare and so some form of observer or state estimator is almost inevitable, and the implementation of these observers and estimators requires recursive filters which are often quite complex. Therefore, the reasons discussed previously for sampling quickly in classical systems also apply to ‘modern’ formulations of control, including other techniques such as H\textsubscript{\infty} design.
The digital controller is basically a set of equations, which forms an algorithm, programmed as a set of instructions on a processor. Within the algorithm, it is widely recognised that the most time consuming operation is the multiplication between the variables and the coefficients. To meet the constraints on the sample frequency, discussed earlier, the processor has to be able to perform the multiplication operation quickly. Various other factors also need to be considered when choosing the processor and these will be discussed in the overview presented in Chapter 2.

As was seen, briefly in section 1.1, the processor choices are either fixed point or floating point in general. Current processors available to perform the task are microprocessors like the 8085 from Intel and the 6502 from Motorola, microcontrollers like the 8051 from Intel (all fixed point), fixed point DSP's like the TMS320 series from TI, the 56000 from Motorola, parallel processors like the Transputer and floating point DSPs, such as the TMS320C30 family. Of these, the processor which can be used in an application will (largely) be determined by the requirement to complete all the required computations within the maximum sample period that will be allowed to achieve a suitable performance.

The processors which can meet the sample frequency requirement will be the most suitable for the application. However, the architecture for the above conventional processors may not have the ability to perform the operations, to the full precision needed for control system processing, at the device's basic clock rate. In most cases macros or subroutines have to be written in order to obtain the required precision, and this means that the operations are not carried out at the basic clock rate. The processor categories mentioned above are discussed in further detail in Chapter 2.

Amongst the alternative approaches available to the above is the concept of direct implementation of the control law on full or semi-custom hardware. That is, a non-programmable processor. However, this would be specific for the control law and
would not be of much use as a general purpose control processor. Hence this concept will not be explored further.

A final alternative is to design a special purpose processor specific for control applications. If a commonality between the Modern control controller structures and the Classical control controller structures can be identified, then a set of consistent equations can be produced. This set could therefore be used as a basis to identify architectural requirements for a Control System Processor (CSP).

The CSP, which is the subject of this thesis, is one such special purpose processor concept whose architecture has been streamlined to meet the needs of real time processing for high performance control systems. The architecture will be derived by taking advantage of the numerical supremacy of the operator such as the low coefficient sensitivity and shorter wordlength requirements for both coefficients and variables.

The high numerical precision required for the internal variables and coefficients can be designed into the processor and hence the high sample frequencies, which can be considerably in excess of the control system bandwidth, can be achieved. This can lead to a very high system performance, due to the low sample and computational delays, and the performance can be comparable to the delay-free computation given by analogue controllers.

This architecture would enable one to implement a wide variety of controllers over a range of sampling frequencies with the resulting equations being consistent for each controller. Additionally, the architecture is such that all the arithmetical operations (add, subtract and multiply) are carried out at the device’s basic clock rate of 15-20 MHz to the full precision needed for control system processing. This will mean that very short computation times and, hence, high sample frequencies can be achieved.


1.5 Summary

With the availability of low cost microprocessors, the trend for using digital controllers is increasing in industry. Most of the controllers are either in transfer function form or in a state space form and it is therefore important to understand:

- the commonality between different controller configurations (transfer function vs state space) to identify if consistent resulting equations can be produced;
- the implementation requirements for these controllers, in terms of structures;
- wordlength requirements for the coefficients and variables;
- the impact of fixed-point number formats on the controllers;
- study the methodologies that are in place to ease the engineers' task of implementation and
- architectural requirements for efficiently processing the digital controller algorithms.

The knowledge gained will further the understanding of the engineer and ease the task of implementation of a digital controller.

1.6 Organisation of the Thesis

The next step in the thesis is to look at the overview which will present a clear picture of the field under consideration and hence identify the main current methodology that can be followed to produce an adequate controller. This will lead, logically, to the remainder of the thesis whereby the deficiencies of this methodology will be discussed and a new method will be suggested to overcome those problems. The remainder of the thesis will examine this proposed solution and evaluate it as a robust technique for implementing digital controllers. The structure of the thesis to address the above is as follows:
Chapter 3 looks at the alternative of realising an optimum structure which will be robust to changes in coefficients and have internal variables with minimum wordlength requirements. This attempt to produce an optimum structure leads to the cross coupled modified canonic structure. Within this context an examination of the internal variable wordlength determination procedures will also be made: namely - round-off noise analysis, deterministic methods such as those used by [Forsythe and Goodall (1991)], and a method based on pseudo-random binary sequences (PRBS). The limitations of each will also be discussed briefly, and the final result will be a thorough understanding of the wordlength requirements.

The literature survey has produced only work being done on optimum structures as far as state space controllers are concerned. Hence Chapter 4 examines the idea of applying the modified canonic structure concept, examined in Chapter 3, to state space controllers. An algorithm is derived which can be used to convert a SISO/MISO state space controller in z-operator form to an equivalent d-operator representation with a modified canonic structure. The limitation of the algorithm is identified, and a procedure to perform sensitivity analysis on a state space controller is derived. The sensitivity analysis performed is an extension of the fractional sensitivity performed on discrete vs continuous coefficients by [Goodall (1992)]. A similar method is derived to assess the sensitivity of the magnitude of the state space system to changes in coefficients.

The interest in architectures for control processors has been increasing since 1990. This is reflected in the literature survey carried out earlier and none of the studies have been for a general purpose processor which can cater for a range of control applications. Chapter 5 therefore looks at the issue of implementation requirements of digital controllers based on the modified canonic structure, and a generic architecture for a Control System Processor is then derived. The issues addressed are coefficient and internal variable formats. A pseudo-floating point format for the coefficients is proposed while the internal variables will have a fixed-point format. Also considered are the instruction set for the CSP and its encoding format, and the core architecture of the arithmetic unit of the CSP -
where issues like registers and multipliers requirements are examined.

Chapter 6 is concerned with the construction and implementation of the ideas presented in Chapter 5. A VHDL model of the processor is described and a test bench is also constructed in VHDL to test the CSP concept. A simple first order phase advance controller is implemented on this model and the results are compared with those obtained from a simulation of the controller in MATLAB to show that the results from the VHDL simulation match those of an equivalent MATLAB simulation.

1 The honour of building the first real arithmetical calculator, one which performed an algorithm automatically, goes to French philosopher Blaise Pascal, whose design of 1642 handled addition and subtraction. This was later extended by Gottfried Leibniz to handle multiplication.

2 Even though this are all taken from the popular press, it makes interesting reading and in some cases it is mind boggling and the imagined scenarios can be frightening.
In order to accomplish the objectives outlined at the end of Chapter 1, one needs to understand the following:

- design of controllers,
- types of controllers,
- requirements of controllers for implementation.

The overview will therefore consider the different design methods that can be used and the types of the resulting controllers. A brief discussion on the different options available for implementation and the issues that need to be considered before an implementable controller can be produced will follow. The issues will include detail on coefficient sensitivity, internal variable requirements, hardware and software considerations. This will be followed by a discussion on the areas within the field of interest that need further investigation and this will, henceforth, provide a motivation for the current research. In order to understand the above concepts, a brief review of the modelling fundamentals of dynamic systems is required; hence this is where the overview will commence.

2.1 Modelling of Dynamic Systems

The majority of dynamic systems are modelled using linear ordinary differential equations. A linear differential equation is one in which the variables and their derivatives have a proportional contribution. Linear relationships are therefore
Overview of the Field of Interest

assumed when deriving model equations. It should however be noted that all physical systems are non-linear to some extent but since it is common to design linear control systems and apply them to non-linear plants, linear time invariant (LTI) systems are the main focus.

2.1 Transfer function models

Differential equations are, however, not the ideal way to represent dynamic systems, as the input, output, and their derivatives are spread throughout the equation. This makes it tedious to determine the relationship between input and output. In addition, when these equations are combined into more complex systems, the mathematics can be considerable.

An easier, more manageable way to overcome the problem is to use the Laplace operator. The resulting mathematics is solved using simple algebra and the result is a transfer function. The transfer function of a system is defined as the Laplace transform of the output divided by the Laplace transform of the input with zero initial conditions. The algebraic nature makes it easier to handle complex systems and a suitable form can be realised. When the inverse transform is applied, the time domain response of the system to the input can be found. A typical nth order strictly proper transfer function is:

\[ G(s) = \frac{Y(s)}{X(s)} = \frac{b_0 + b_1 s + \ldots + b_m s^m}{a_0 + a_1 s + \ldots + a_n s^n} \quad (m \leq n) \]  

(2.1)

2.1.2 State Variable Models

State variable models are based on describing the system dynamics as a set of simultaneous first order linear differential equations called the state equations. The state equations are expressed in standard matrix vector form and much of the analysis and design is done using matrix manipulation techniques.
The fundamental concept behind the state variable models is that the dynamic condition of a system at any instant is completely described by its state. The state is defined in terms of a set of state variables, \( x_1(t), x_2(t), \ldots, x_n(t) \). Knowledge of these state variables together with any inputs allows the future state of the system to be found from the state equation. The system output is expressed in terms of the state variables and the inputs. An nth order system requires n state variables and n state equations to model its dynamics. The state variable model is of the form

\[
\begin{align*}
\dot{x}(t) &= Ax(t) + Bu(t) \\
y(t) &= Cx(t) + Du(t)
\end{align*}
\]

where

- \( x \) is a \( n \times 1 \) state vector, 
- \( A \) is a \( n \times n \) coefficient matrix,
- \( u \) is the \( r \times 1 \) input vector (for \( r \) inputs), 
- \( B \) is a \( n \times r \) coefficient matrix,
- \( y \) is a \( p \times 1 \) output vector (for \( p \) outputs), 
- \( C \) is a \( p \times n \) coefficient matrix,
- \( D \) is a \( p \times r \) coefficient matrix.

Note that a transfer function of the model can be obtained from the state variable representation. It can be easily shown that equation (2.2) gives:

\[
Y(s) = [C(sI - A)^{-1}B + D]U(s)
\]

(2.3)

It is also possible to formulate the state variable representation from a transfer function.

### 2.1.3 Performance Specifications

Once the dynamic system has been modelled, specifications on the performance of the system to various inputs can then be set using either the system time or frequency responses. If the time response is used, then targets such as the percentage (or peak) overshoot, decay ratio, settling time, response time etc. can be specified. If the frequency response is used, then another set of parameters such as the gain & phase margins, bandwidth of the system etc. can be specified such that
the system is always stable. The control system can then be designed with the aim of achieving these targets. Implementation can then follow.

2.2 Design Methods

Control engineers have at their disposal a number of different tools for achieving the target specifications. Broadly speaking, the controller design can be performed using either the classical or modern control techniques. The classical techniques date back to the 1930's while the modern control techniques date back to the 1960's. The 1980's has seen the emergence of other methods such as $H_\infty$ control, the use of fuzzy logic and neural networks to perform control functions.

2.2.1 Classical Control

In classical control methods, the physical system is usually represented as a transfer function and the design can be done in the frequency domain or on a root locus by placing a compensator in the forward path of the control system. The design can also be done in the $s/z$ plane since there is a direct relationship between the $s/z$ plane pole and zero positions and the time domain behaviour. A root locus diagram is constructed and used as a design aid to construct the controller. Once again, a compensator is the result.

Example compensation stages are Phase Lag, Phase Lead, PID. The resulting controller is usually a transfer function. If the design is done in the $s$-plane, the resulting controller is known as a continuous controller while the $z$-plane design method results in a discrete controller. Classical control techniques can be extended to MIMO systems, in which case a transfer function matrix description results.

Amongst the modern control techniques dating post 1960, some of the common examples are as described in the following subsections.
2.2.2 Optimal Control

Optimal control methods [Kwakernaak and Sivan (1972)] use the state variable representation to describe the physical system and the controller is designed, usually, in the time domain resulting in feedback gain matrices designed using full state feedback. However, access to all the states is rare and hence some sort of observer or state estimator has to be designed. The resulting controller can be either a continuous or discrete controller depending on which time domain was used. The controller is usually described in a matrix form similar to the one given in equation (2.2).

2.2.3 Robust Control (H∞ etc.)

Recently, the emergence of Robust Control techniques which formally accommodate system uncertainty, e.g. $H_\infty$ control, has sparked a great deal of interest [Doyle et. al. (1989), Glover and Doyle (1990), Doyle et. al. (1992)]. This is a frequency domain based method but uses the state space representation to describe the system and the controller is a transfer function matrix which can also be expressed in the state space form given in equation (2.2). It is a very powerful technique if the design is performed with care, but a downside is that the resulting controller is often of the same, if not higher than, order as the physical system under control. This makes it more difficult to implement the controller.

2.2.4 Unified Theory Approach

Amongst the most interesting methods available is the unified theory approach to digital control and estimation [Middleton and Goodwin (1990)]. The work is based on the divided difference operator, also known as the $\delta$-operator in this thesis. The foundations of the work lay in use of the divided difference operator to produce digital controllers that were robust to quantisation effects and had improved wordlength requirements in both the coefficients and variables. This ultimately led
to the formulation of a unified theory for continuous and discrete time methods giving a better understanding of discrete-time control under fast sampling.

When the dynamics of the sampled data system are represented by this operator, as opposed to the conventional z- (or shift) operator, it leads to a novel systems calculus that allows for a unification of continuous and discrete time formulations, thereby enabling smooth transition from a discrete (sampled data) algorithm to its continuous time counterpart. This consequently enhances the numerical conditioning of the algorithms in the high-speed regime. The calculus that is formulated, based on the divided difference operator, is used as a framework to treat problems on system state estimation, system identification and control system design. The resulting controller is numerically superior to an equivalent z-operator based one. The application of this approach to signal processing has also been discussed [Goodwin et al (1992)].

2.2.5 Other Methods

Fuzzy Logic

Application of Fuzzy logic to controllers has also become popular in recent years even though the concept of fuzzy logic was introduced well over 25 years ago. The early results provided a formal theory of qualitative descriptions and procedures which brought mathematical precision to a type of uncertainty distinct from probability [Zadeh (1965)]. Contrary to popular belief, fuzzy logic is completely deterministic and result is a non-linear controller which is robust and effective for a wide range of applications. The method is based on many-valued logic which enables general principles and expert knowledge to be used to provide control rules and procedures [Johnston (1994)]. The technique was first demonstrated on a steam engine [Mamdani (1976)] and soon found applications in complex industrial plants, such as cement kilns [Ostergaard (1977)], and slow processes, such as sewage treatment [Tong et al (1980)]. The advantages that could be gained over conventional PID controllers was best demonstrated by the example of the Sendai
urban railway [Yasunobu and Miyamoto (1985)]: these were smooth adjustment to varying loads, improved ride quality and an improved fuel economy (the fuel consumption was reduced by 20%).

**Neural Networks**

Neural networks are another form of non-linear controllers which are increasingly being used in the design of control systems either directly or indirectly. For example, the control of copper lasers [Buckley and Richardson (1996)] is an active on-line process where multiple coupled neural networks are used to first identify the load conditions and then alter the load conditions to minimise the stress placed on the control circuitry and hence increase the lifetime of expensive components. Meanwhile, the tuning of PID controllers, “the most common controller in industry”, is an example of an off-line design process [Ruano et al (1991)] where a neural network is used to automate the process of tuning the PID parameters for a particular plant. Further examples of the use of neural networks in Control can be found in [Warwick et al (1992)].

**2.3 Types of Controllers**

Due to the finite amount of time available for the research it was decided that fuzzy logic/neural network type of controllers would not be considered as it would have resulted in a subject area that would be vast and unmanageable. This thereby reduces the area to linear time invariant (LTI) controllers that are represented as either transfer functions or in the state variable format. Additionally, it has been seen that it is possible to get a transfer function representation from a state variable one (equation 2.3) and vice versa. The respective controllers can be represented either in discrete or continuous form.
2.4 Implementation

As has been seen, the implementation can be carried out using either analogue or digital techniques. If the implementation is to be digital, a number of factors need to be considered. If the resulting controller is continuous it will be necessary to convert it into discrete form, something which is not necessary of course if the design process yields a discrete controller directly. Either way, the next step would be choose an appropriate implementation structure and derive wordlengths for the coefficients and the internal variables.

2.4.1 Discretization

If the resulting controller is continuous then a discrete equivalent needs to be derived. This is usually done through the process of discretization. It has been argued that this is an inefficient way of deriving the discrete controller. The inefficiency is especially evident when considers that the sample frequency typically needs to be in excess of 30 times the dominant controller frequency; only then can one produce a discrete controller which performs on a par to the equivalent continuous controller. Additionally, the discretization process means that one has to give up some of the possibilities that are present only in discrete design (e.g. deadbeat behaviour). Finally, the resulting discrete controller is simply imprecise because the discretized control never behaves like the continuous design. In practice, however, experience has shown that none of these arguments have a significant relevance and there may be several reasons why the indirect way via continuous design may be a better choice [Hanselmann (1987)].

Discretization is usually done through *Emulation* and any of a number of transforms can be used but only the *bilinear transform* is considered here [Golden and Kaiser (1964)]. The bilinear transform is also known as Tustin’s method and essentially gives trapezoidal integration. It has the desirable property of never generating unstable z-domain poles as long as the s-poles are stable. Another property is that
the frequency response of the continuous transfer function is approximated in the frequency response of the discrete system, albeit with a warped frequency axis. The bilinear transform is given by:

\[ s = \frac{2z^{-1}}{Tz+1} \]  

(2.4)

The bilinear transform is widely in use, and tests on numerical examples indicate that it is a good choice [Katz (1981)]. It is also quite simple to formulate this method in state space form for multivariable systems. The discrete form of equation (2.2) can be derived by substituting the bilinear transform and can be found [Haberland and Rao (1973)]. Other transforms methods also exist to perform the emulation and examples can be found in [Katz (1981) and Franklin and Powell (1980)].

Discretization does have a penalty on the controllers. The zero order hole (ZOH) at the output of the controller introduces a phase lag, and therefore the frequency response of the discretized controller is likely either to show more negative phase compared to the continuous controller, or to show an increased gain in the higher frequency region. Therefore stability and damping problems could occur. To preserve the behaviour of the continuous system, the sampling frequency has to be at least 10 times higher than the crossover frequency [Hanselmann (1987)]. It is possible to take the phase lag of the ZOH into account during the discretization process, but the price of delay compensation is an increased high frequency gain.

2.4.2 Structures

Once the discrete controller is known, the next step of implementation is to choose a structure for the controller. The ideal structure should have the following properties:

- minimum possible number of storage elements - implying minimum memory needed
- minimum number of non-zero and non-unity coefficients - thereby reducing the number of multiplications that need to be performed.
- minimum computational delay - thereby minimising phase lag and hence instability
Overview of the Field of Interest

- multi input/output capability - therefore a wide range of controllers can be implemented from SISO to SIMO to MIMO
- state space description should be possible - hence making the structure useful for controllers represented in both transfer function and state space forms
- minimum coefficient sensitivity - which leads to shorter wordlengths
- minimum internal variable wordlength requirements.

Before looking at the different structure alternatives, consider first the issue of wordlength requirements for the structures.

2.4.2.1 Wordlength Requirements

For any structure, the wordlength requirements can be classified into coefficient and internal variable requirements. More often than not, the choice of the processor on which the controller is implemented has been used to determine the wordlength of the internal computations (that is, coefficients and internal variables). This has the wrong emphasis as the issue that needs to be addressed is not ‘How will the performance be affected if the coefficients or internal variables are quantised to a given precision?’ but rather ‘To what precision must the coefficients or internal variables be quantised to achieve a certain performance?’.

The number of bits required for the coefficients is a function of the sensitivity of the structure, while the number of bits required for the internal variables is, as will be seen later, dependent on three factors: the accuracy to which the digitised controller output should follow the ideal output based on an infinite wordlength, the resolution of the input/output data and on the maximum gain of the controller.

Coefficients

By forcing the coefficients to a fixed number of bits, the function of the controller is modified compared to what is designed. This is acceptable if the sensitivity of the structure is low, but not for cases where the sensitivity may be relatively high. Due
to the high sensitivity, the system poles are moved into undesirable locations hence modifying the function of the controller which can give rise to instability. Generally, the more sensitive the structure, the higher the number of bits that are required to represent the coefficients to still achieve an acceptable performance from the controller.

A number of different methods are available to analyse the sensitivity and hence calculate the wordlength requirements of the coefficients [Mitra and Sherwood (1974), Knowles and Olcayto (1968)]. The one adopted for this thesis is the fractional sensitivity approach outlined in [Goodall (1992)]. The advantage of this method is that it does not include zero and unity coefficients in the analysis. From a practical point of view it is helpful to think in terms of percentage accuracy rather than absolute sensitivity which means no recognition of the actual size of the coefficient; this is where the main advantage of Goodall's method lies. Additionally, as will be shown in section 2.4.2.4, the inclusion of zeros and unity coefficients is not given sufficient attention in the derivation of sensitivity measures.

Internal Variables
The internal variables of a controller are the intermediate results that occur during the computation of the output. This will be clearer when structures are examined in section 2.4.2.2. The total number of bits required for these internal variables is given by the sum of three terms:

1/ The number of bits used for the input and output data, the resolution of which is chosen to meet the system accuracy requirements;

2/ the number of bits required to account for underflow. These bits are necessary to ensure that the effect of small input values correctly propagates through the structure and is not simply truncated to zero when multiplied by small coefficients. The number of underflow bits can be derived from the structure in terms of the coefficients and the fractional output accuracy for small inputs [Forsythe and Goodall (1991)]. Further details will be discussed in
Chapter 3 where the other methods that can be used to determine the number of underflow bits will also be considered;

3/ the number of bits required for internal overflow. These are required to ensure that the internal states and intermediate results do not saturate to maximum input values. In this case the output will saturate (and this is unavoidable), but the internal variables must not saturate such that recovery from the situation is swift once the offending input is removed. The number of overflow bits can be determined from the maximum overall gain that is likely to be seen in practice.

2.4.2.2 z-operator

Assuming for now that transfer functions are the starting point, there may be seemingly natural choices for obtaining programmable difference equations. For example, a strictly proper SISO controller has the form

\[ G(z) = \frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}}{1 + a_1 z^{-1} + \ldots + a_n z^{-n}} \quad (m \leq n) \] (2.5)

which can easily be implemented as

\[ y_k = -a_1 y_{k-1} - \ldots - a_n y_{k-n} + b_0 u_k + \ldots + b_m u_{k-m} \] (2.6)

However, this is only the simplest possible equation and it requires more storage elements than necessary. Other more efficient structures exist and are given in [Phillips and Nagle (1984)]. Of the many efficient different structures that result, the thesis will concentrate on the canonic form, shown in Figure 2.1, as it satisfies a large number of the properties discussed at the beginning of section 2.4.2. Since any higher order function can be implemented as a combination of first and second order blocks either in cascade or parallel form, only second order structures will be considered unless otherwise stated.
The following difference equations define the above structure:

\[ v_k = u_k - a_1 v_{k-1} - a_2 v_{k-2} \]
\[ y_k = b_0 v_k + b_1 v_{k-1} + b_2 v_{k-2} \]

where \( v_k = v(kT), v_{k-1} = v(k-1)T, \) etc.

And this can be implemented in software (real-time) using the following equations (in order given), for \( n=2 \):

\[ v_0 = u - a_1 v_1 - a_2 v_2 \]
\[ y = b_0 v_0 + b_1 v_1 + b_2 v_2 \]
\[ v_2 = v_1 \]
\[ v_1 = v_0 \]

where \( v_0, v_1, \) and \( v_2 \) are the internal variables discussed in section 2.4.2.1. The above structure is based on the z-operator, and it has long been recognised that there are sensitivity problems associated with the z-operator based structures [Liu (1971)] - particularly with respect to the inevitable approximations in their coefficients. This case is particularly severe in recursive filters in which the sample frequency is considerably higher than the highest system frequency of the transfer function to be implemented. These problems are very significant when designing digital controllers for high-speed, high-performance control systems, in which it may be necessary to sample quickly in order to minimise the effect of sample delays.
Overview of the Field of Interest

[Goodall (1990)]. Fortunately a remedy to this case exists.

2.4.2.3 δ-operator

It involves the use of "delay replacement" operator also known as the δ-operator [Goodwin (1985), Goodall and Brown (1985)]. By making the following substitution

\[ z = \delta + 1 \]  

(2.8)

in equation (2.5), it can be shown that \( G(z) \) becomes

\[
G(\delta) = \frac{Y(\delta)}{U(\delta)} = \frac{c_0 + c_1 \delta^{-1} + \ldots + c_m \delta^{-m}}{1 + r_1 \delta^{-1} + \ldots + r_n \delta^{-n}} \quad (m \leq n)
\]

(2.9)

Note that the definition of Goodwin is \( \delta = (z-1)/T_s \) where \( T_s \) is the sample interval but the principle is identical. Note that in our case where the definition of equation 2.8 is used, the \( T_s \) usually appears as a multiplier in the \( d_i \) and \( d_s \) coefficients and hence the resulting answers are similar to those that would be obtained if one were to use the definition of Goodwin.

The \( z^i \) blocks are replaced by the \( \delta^i \) blocks and the coefficient sensitivity of the structure is considerably reduced. As the implementation of the \( z^i \) block involves a shift operation, the implementation of the \( \delta^i \) block involves a shift and an add operation. That is,

\[
\alpha_k = \delta^i \beta_k = \alpha_{k,i} + \beta_{k,i}
\]

(2.10)

where \( \alpha \) is the output variable, \( \beta \) is the input variable to the \( \delta^i \) block.

The improvements possible from this approach were also identified a decade earlier by [Agarwal and Burrus (1975)], but the idea of a completely new operator was not developed at that time. Recently the interest in the use of δ-operator systems has increased such that various researchers are publishing material on the uses and some of the shortcomings of this method. A typical representative example is the
work done by [Premaratne et al (1994)] to formulate discrete time equivalents of continuous time systems directly in δ-operator representation rather than first going to the z-operator system and then using equation (2.8) to obtain the equivalent δ-operator system. Another example is the work of [Chotai et. al. (1991)] which applies the δ-operator to the non-minimum state space approach to True Digital Control (TDC). In this case the designs that result are found to be normally more appropriate than the z-operator designs especially when the sampling frequency is chosen to be high in relation to the dominant time constants of the controlled system. As a final example of the increasingly wide spread use of the δ-operator is the work of [Tadjine et al (1992)] to describe the LQG/LTR method using the δ-operator.

As with the z-operator, there is a large number of possibilities for the structures using the δ-operator and once again only the canonic form will be considered. The δ-operator canonic structure can be further modified such that an uniformity in the scaling of the internal variables be realised. This is achieved by moving the feedback coefficients into the forward path to obtain the modified canonic structure [Goodall and Brown (1985)]. An example of a second order structure is given in Figure 2.2.

This is implemented by the following equations (in order given) in the δ-domain:

\[
\begin{align*}
    v &= u - w - x \\
    y &= p v + q w + r x \\
    x &= x + d_2 w \\
    w &= w + d_1 v
\end{align*}
\]  

(2.11)

It is on this structure that the thesis will concentrate on during the remaining chapters. The structure has been shown to be highly robust to coefficient sensitivity and also requires fewer bits to represent its internal variables than an equivalent z-operator based canonic structure of Figure 2.1. It is a 'known to be good' structure. However, one further class of structures will be considered in the next section: these are known as optimum structures.
2.4.2.4 Optimum Structures

So far the structures that have been considered have been with respect to controllers being represented as transfer functions. For controllers represented in the state space format, the work published in the literature has concentrated on the optimum structures concept.

Optimum structures were first known as minimum round-off noise structures and were first proposed by [Mullis and Roberts (1976)] and [Hwang (1977)] to address issues in the signal processing/digital filtering field. This work has since been extended by a number of researchers including [Thiele (1984)] and [Li and Gevers (1990)] with the application of these optimum structures to closed loop control as the prime objective. Additionally, the work of Li and Gevers has looked at the optimum structures for both $z$- and $\delta$-operators [Li and Gevers (1993)].

The early work was based on minimising the signal quantization noise arising from the update of the states while still retaining the scaling of the state vectors. Scaling of the states is performed such that the overflow probability is made equal for every state variable assuming a white noise input signal. The recent work is essentially an
extension of the earlier work and involves the simultaneous minimisation of coefficient sensitivity and the noise introduced due to the rounding off of the computations during a state update and output calculation operations.

It is argued that due to the finite nature of the problem, a finite number of different controller realisations are possible. By defining a sensitivity measure [Tavsanoglu and Thiele (1984)], a set of necessary and sufficient conditions for an optimal state space realisation are given, and from these conditions a simple explicit minimisation procedure is derived.

The end result is a transformation matrix, $T$, which when applied to the original controller, produces an optimal state space system which is both robust to coefficient changes and has minimum round off noise due to computations. The concept looks very sound but further examination has revealed two important points. These are worth bearing in mind if one wishes to use these optimum structures:

1/ The sensitivity measure, as proposed in [Tavsanoglu and Thiele (1984)] and used to derive the optimal structures, has a limitation. Closer examination will show that the measure does not know whether any of the individual coefficients are zero or one. Hence it treats them all in the same manner and thereby perturbs the ones and the zeros. This can be a problem because in reality a one is never implemented as a multiplier coefficient and instead an addition is usually performed. Also, it is possible to get a system where some of the zero coefficients may be very sensitive to perturbations. Hence, in a measure, the effect of sensitive zero coefficients should not appear. Ideally, the sensitivity measure should be such that it does not consider the zeros and ones that may appear in the system.

2/ The optimal realisations also suffer from the fact that the $A$ matrix of the controller generally has no specific structure. All coefficients can be non-zero and non-unity. Hence, it places a large computational overhead on the processor, particularly as the order of the controller increases, and can make
Overview of the Field of Interest

it difficult to achieve the necessary sample rate. One may have to resort to a faster than necessary processor to achieve the objective. There may, however, be cases where it might be more efficient to use an optimal structure. For example, if such a structure enabled all the computations to be done using single word arithmetic while the sparse arithmetic required multi-word operations, then the former might lead to a faster solution.

Now that the structures have been considered one can begin to appreciate the wordlength requirements in terms of coefficients and internal variables. It is important to have procedures which are practical and reliable to use. A practical method to calculate the wordlengths of the coefficient's has been described in [Goodall (1992)] while for internal variables, the methods that exist are described and explored in detail in Chapter 3. The wordlengths of the δ-operator structures along with their formats are discussed in further detail in Chapter 5. These points should be borne in mind when looking at the requirements of any hardware being considered for implementation.

2.4.3 Hardware Alternatives

The next stage in the design process involves decisions regarding choice of processor; that is which processor should be used to accomplish the task at hand. As has been seen, the microprocessors that can be used to implement the controller can be classified into two main categories: floating-point and fixed-point. The number of alternatives that exist for implementing the algorithms is vast. The range is as given below:

1/ At the upper end there exist special machines for rapid experimenting such as the ADIO system [Powers (1985), Kerckhoffs et al (1985)], which have a very large on-cost.
2/ **Word slice** chip sets which can be used as building blocks to develop microprogrammed high-speed signal processing systems are at the next level. In this subclass, there are special purpose arithmetic chips, solely for accumulating/multiplying floating point numbers. Control of these devices must be derived from microcode memory and control logic.

3/ **Microprocessors** form the third subclass. Their advantage to the above is *ease of implementation* and *testing* of the controllers. The end result is achieved at a much lower cost and effort. The systems from these processors are easy to program in a high level language *but* the speed attainable is only *medium* because they do not have a built in hardware multiplier. The multiplication is usually performed in software using a built in multiply instruction, which will be microprogrammed, but this is usually a 8x8 bit or 16x16 bit multiplication which is not accurate enough. This often results in a controller architecture that is not fast enough for the control of 'fast' mechanical systems such as automotive engines and electronic systems such as the control of high frequency PWM converters [Holme (1994)]. The execution speed of the controllers can be increased by attaching fast hardware multipliers at the expense of extra effort but this is hardly worth doing. More sophisticated microprocessors with floating point units - such as the 80486 from Intel - are now available but are not commercially viable to use in control applications, as the cost of purchasing the processor is too great. The big drawback for microprocessors is that their architecture is more useful for data rather than numerical processing. They do have a large address space at their disposal but this is usually not a requirement for a control implementation problem.

4/ **Microcontrollers** are usually classed as microprocessors with all the necessary control specific peripherals. Typical on-chip features include A-D converters, timers and PWM outputs. This class of processors is suited to applications where the actual control algorithm is only one of many other tasks such as sequencing, complex timing, and communication. The
computing speed achievable is again 'medium at best'. An example of a modern microcontroller is the Intel 80196 which, unlike the Intel 8051, includes a hardware multiplier. The functional units included on a microcontroller are however quite useful and reduce the chip count on a PCB considerably.

5/ Digital Signal Processors (DSP's) became popular with the introduction of the TMS320C10 by TI in 1982. It was the first processor which had been designed with numerical processing as a target, and the application it was aimed for was communication signal processing. The power it achieved had nothing to do with any exotic silicon process technology but due to the architecture. The key features are:

- the Harvard architecture, which has separate buses for program and data. This removed the intrinsic bottleneck of the Von Neumann architecture. In the Von Neumann architecture, the strict sequentiality means that the data can never be processed faster than its accompanying control;
- and the inclusion of an on-chip hardware multiplier to perform the time consuming multiplication operation. This reduced the execution time of the algorithm considerably.

Sophisticated control strategies were soon being implemented using these DSPs. The saving grace of the DSPs is the sheer speed at which they run, but nevertheless it is not straightforward to design and code the computational software for control processing. The attractive computation speed meant that non-trivial controllers which required high sampling rates could be implemented.

6/ The need for high sample rates and complex high order controllers has increased the need for processors with powerful computing engines such as the INMOS Transputer, a parallel processor. The Transputer has been
applied to a wide range of problems including digital flight controllers, field-oriented control of AC induction motors and the inverse dynamics problem in robotics. The work has been reported in the literature [Fleming (1988)]. Although transputers are suitable as computing engines they introduce additional difficulties such as organising of the tasks on the different processors and the code design complexity invariably increases.

7/ Floating Point processors can also be used to implement controllers. Computationally, they are adequate for the task at hand but are not optimised for control and are inevitably less efficient (due to expense, increased package size, and power consumption).

8/ The final alternative is based on the idea of designing a semi- or full-custom made processor suitable for control implementation. This alternative has become increasingly attractive in recent years due to the advances made in silicon technology. This has meant that it is possible to design processors whereby the architecture was tailored to address the needs for a controller. For example, the wordlengths requirements of the internal variables and the representation of the coefficients could be specified as part of the design process.

As was seen earlier, a 16 bit wordlength for the internal variables is rarely sufficient for control (even though input/output variables of 10 or 12 bits are common), and so higher precision is necessary. With the δ-operator structures, high coefficient accuracy is not necessary and the coefficient can be represented in a simple pseudo-floating point format, with a number of significant bits and a simple exponent; further details are given in Chapter 5. It is therefore possible to depart from the norm of equal wordlengths for the operands of the multiplication instruction. The hardware multiplier could be designed with the knowledge that structures using 24 bit variables multiplied by 4 or 6 significant bits of coefficient to yield a 24 bit result have been found to be very effective. Therefore, for any processor to be efficient, it
must satisfy the above requirements. A processor as such can be designed using Application Specific Integrated Circuit (ASIC) technology. This will be described in further detail in the next section.

2.5 ASIC’s and CSP’s

The rapid development of silicon technology has made ASICs a more popular way of realising signal processing algorithms in general. The systems can be implemented easily as only a few basic operations, such as additions, delays and multiplications, are needed. The application base is quite large and examples of custom built ICs can be found in video image processing [Kwentus et al (1992)] to signal processing [Woods and McCanny (1992)] to control [Burge et al (1994)], with the majority of applications still in digital signal processing. This is a far cry from the mid to late 1970’s when special purpose hardware was being designed to perform real-time processing using MSI and LSI technology [Bin Nun and Woodward (1976)], and bit-slice micro-processors were being proposed in filter implementation technology [Woodward (1979)].

In the control field, various proposals have been made by researchers and engineers using novel hardware for controller implementation [Lang (1984), Jaswa et al (1985), Breitzman (1985), Jover and Kailath (1986), Jacklin et al (1986)]. However, none of the above proposals was for a general purpose control system processor which would cater for a range of control applications. A brief discussion of the above alternatives is given below followed by a look at other proposals that have recently been made.

[Lang (1984)] proposed the Digital Control Processor (DCP), which was targeted at a linear-regulator control system; it used logarithmic data representation and arithmetic, resulting in compression of data and high-speed arithmetic (especially multiplication). The DCP was intended for stand-alone operation and had no...
provision for interprocessor communication. It was only intended as a design study and no hardware was built.

[Jaswa et al (1985)] proposed a concurrent processor architecture for control (CPAC) which was intended to function as a coprocessor closing the loop around a controlled plant. Two main elements were proposed: a continuous and a data processing element (CPE and DPE respectively). The physical separation of the continuous and discrete components of the real-time controller exploits the existing concurrency between these operations. The CPE is optimised for continuous control and the state space structure of the control algorithm. The DPE is optimised for discrete control and is the decision making element of the CPAC. It remains in overall control of the CPE. Again, the CPAC did not develop beyond the design study stage - no hardware was developed.

[Breitzman (1985)] describes a novel architecture for control which was designed and built in close conjunction between Ford and Intel. The description is of a two-chip set which Ford uses for the specific task of automotive emissions control and fuel economy. This chip set is now also used for anti-lock braking systems (ABS) and active control of vehicle response under certain conditions. One chip is a 16-bit microprocessor with an ADC and a DAC while the other chip is a mixture of RAM and ROM. It is with this chip set that Intel went on to produce the 8061.

[Uover and Kailath (1986)] discuss the potential of Kalman Filter algorithms on systolic arrays, while [Jacklin et al (1986)] shows how an array processor can be used to speed up complex control calculations.

The interest in architectures for control has been increasing since 1990. Examples are the PACE project, which looked at automatic mapping of block diagrams into arrays [Spray and Jones (1991)] and the ASIC implementation of SISO control laws [Burge et al (1994)]. A major international workshop on architectures for control has also been held where systolic arrays and proposals for specialist hardware implementations were presented, with increasing support for hardware
architectures for control systems [Xu et al (1991), Fleming and Jones (1992)]. Some of the presentations at the workshop and other examples are discussed further below:

[Halang et al (1991)] describes the implementation of a real-time variable structure controller on a chip using a minimum RISC architecture. The controller’s applications vary from servo regulators to manoeuvrability control of spacecraft and VSTOL aircraft, to the control of interconnected synchronous machines. The chip is meant to be a stand alone chip, and the principle is to have minimum possible components on chip. The architecture is such that it would be suitable for performing the variable structure control algorithm described within. There is no evidence of any hardware having been built.

[Aude and Aude (1991)] describe a hardware accelerator for a robot arm using multivariable self tuning control. The hardware accelerator has been designed to achieve two main objectives: provide the necessary processing power required to implement the complex self-tuning control algorithms, and to be flexible enough to ensure it can be used in the implementation of different control algorithms in conjunction with different control applications (as opposed to robot arm control). The accelerator is meant to work in conjunction with a host computer such as any IBM PC compatible microcomputer, and has effectively a number of identical processing elements connected over a shared bus, sharing dual-port memory through which they communicate with the host computer. The processing elements have: a 32-bit RISC microprocessor based on the SPARC architecture, an optional floating point co-processor, a ROM, a fast RAM, an optional microprogrammable vector co-processor, and an interface to the accelerator internal bus. Once again, it is not evident if any hardware was built from the study.

[Burge et al (1994)] describe the implementation of SISO control laws on an ASIC. They describe the design of a fixed structure firmware programmable device which will cater for what is referred to as ‘general purpose’ SISO controller. The controller is designed to accept a variety of inputs - such as analogue and digital inputs which
Overview of the Field of Interest

can be in the form of either binary or gray code - and operate on these inputs to produce a control signal output, again in a variety of forms - such as pulse-width modulation (PWM), digital and analogue outputs. It can also support a wide range of sampling frequencies which can be programmed. The PID structure of [Astrom and Wittenmark (1990)] is chosen to be the fixed structure for the device. This gives the capability to implement a wide range of conventional SISO control laws including P, PI, PID, and PDF. The ‘gain’ parameters of the forward and feedback paths can be programmed. The ASIC was designed in 1.5 μm CMOS technology. The device included two multiplexed hardware multipliers with the algorithm architecture already generated to minimise the computational time.

2.6 Summary

Very briefly, some of the issues involved in the design of a digital controller have been examined. The assumption has been that the controller can be designed either in the continuous time domain and then discretized using an appropriate transform such as the bilinear transform, or designed directly in the discrete form. The digital controller then has to be analysed and an appropriate structure chosen, the wordlength requirements (both coefficient and internal variable) of the controller can then be calculated. Once this is done, the processor to implement the algorithm can be chosen subject to it satisfying certain requirements such as sample frequency constraints as discussed earlier in Chapter 1. The above points are clear when the overview is considered as shown in Figure 2.3.
Overview of the Field of Interest

Methods for designing the controller

Design is done either in the Continuous (s) or Discrete (z or $\delta$) domain

Classical Control | Optimal Control | H_∞ | Unified Theory

Classical Control

Optimal Control

$H_\infty$ Theory

Unified Theory

Others (eg Fuzzy/Neural)

Methods for designing the controller

Design is done either in the Continuous (s) or Discrete (z or $\delta$) domain

Classical Control | Optimal Control | $H_\infty$ | Unified Theory

Classical Control

Optimal Control

$H_\infty$ Theory

Unified Theory

Others (eg Fuzzy/Neural)

TF Matrix or Structure

A, B, C, D Matrices

$y = [\begin{bmatrix} TF & TF \\ TF & TF \end{bmatrix}] u$

$x = Ax + Bu$

$y = Cx + Du$

Understand Commonality between TF's and SS

Common structures for TF and SS, and hence common equations, that is easy to implement.

Structures eg $\delta$-operator structure

Understand Wordlength determination routines of internal variables.

Architectures for Processors, eg sequential, parallel, pipelined etc

Processor choice C : S : P

Formats for coefficients and variables, multiplier design, etc

Figure 2.3 - An Overview of the field of interest

The emphasis of the thesis will be on the implementation requirements for these digital controllers, and hence will address issues such as structures, wordlength requirements, architectural requirements for a processor, all of which will be used to implement the controllers in general. These requirements apply regardless of whether the controllers are represented in transfer function or state space format.
Chapter 3

STRUCTURES FOR IMPLEMENTATION

Several different structures can be used to implement a controller, whether the controller is specified (or represented) in transfer function form or state space form. Different structures correspond to different sequences of arithmetic operations; from the numerical point of view the order of the computations within these sequences would be irrelevant if the precision with which one was working to was infinite, although different structures might have different numbers of arithmetic operations which would clearly affect the computation time. However, for digital controller implementation in practice one cannot assume infinite precision.

Hence, quantisation errors of various kinds, such as sampler quantisation error, coefficient representation error and internal variable truncation (or rounding) error, are created at certain points in the arithmetic process. These errors affect the output of the controller and the magnitude of the quantisation error in the output is determined by the order in which the arithmetic operations are performed [Liu (1971)]. The sampler quantisation error is really a system issue in that it affects the input/output resolution and is not directly a processor issue while the other two errors will affect the architectural considerations of the processor. Also, the structures being studied exclusively use the δ-operator, for which it is well proven and accepted that coefficient sensitivity is not a problem. Hence, of all the above errors only the internal variable truncation (or rounding) error will be considered in this chapter.

In this chapter two structures will be studied, the cascaded modified canonic structure and the cross coupled modified canonic structure, both based on the δ-operator, which can be used to implement controllers in transfer function form.
A wordlength analysis for the two structures, in terms of overflow and underflow bits required for fixed-point implementation, will be performed. Two different methods, the steady state analysis and round-off noise analysis, will be used to analyse the underflow requirements. Both methods will be compared and checked by simulation, and it will be shown that the steady state analysis gives a more realistic assessment of wordlength requirements.

The Internal Variables

Before looking at the structures and analysing the wordlengths, it is important to clarify the assumptions that are made regarding the representation of the variables.

The input/output variables will have been defined to have a certain wordlength depending on the resolution of the ADC being used. This will form a basis for the internal variables within a discrete filter section. However, the internal variables must be examined further to determine whether any overflow and underflow bits are necessary. It may not be immediately obvious whether either or both of the requirements are needed or not, but it is nevertheless essential to check and allow for any such requirements, or else the filter will not work properly under all conditions.

Figure 3.1 summarises a general format for the variables that will be used in the analysis carried out in this chapter. The position for the binary point in the variables is arbitrary, and for convenience is chosen such that the input/output variables become integers. The overflow bits in the internal variables allow for 'growth' beyond the maximum size of the input and output variables, and underflow provides fractional bits.
3.2 The Cascaded Modified Canonic Structure

This structure was first reported in [Goodall and Brown (1985)]. It is essentially a canonic structure with unity feedback, achieved by moving the coefficients into the forward path of the filter, with appropriate changes in the coefficients which form the output $y$. Since the structure was first known as the modified canonic structure, the thesis will continue to use this term instead of the cascaded modified canonic structure unless clarity requires the use of the cascaded prefix. The structure for a second order controller is shown in Figure 3.2, and this can be readily extended for higher orders.

[Figure 3.1 - General Number format for variables]

[Figure 3.2 - A second order modified canonic structure]
The modification means that the internal variables $v$, $w$ and $x$ now have maximum values that are of the same order as that of the input variable $u$. That is, scaling has effectively been performed. This is interesting as scaling is often performed prior to implementing fixed-point digital filters to reduce the probability of overflow in the internal variables [Hanselmann (1987)]. This will be examined further when the overflow of the internal variables is considered in section 3.2.1.

A further advantage of the modification is that some of the numerator coefficients can be approximated to 0 or 1 in certain circumstances, particularly at high sample rates, resulting in a reduction in the number of multiplications needed [Goodall and Brown (1985)].

### 3.2.1 Internal Variable Overflow

Overflow produces errors that are unacceptable, which can lead to instability and sufficient care must be taken to prevent them [Li and Gevers (1993)]. One method is to scale the variables [Hwang (1977)] and another is to allow for a number of overflow bits [Forsythe and Goodall (1991)]. The method adopted here is to allow for overflow bits; the analysis that follows looks at the steady-state value of the internal variables in response to a maximum input.

If any of the internal variables $v$, $w$ or $x$ might become larger than the maximum value of the input, particularly in response to a maximum input change, then a number of overflow bits must be provided to allow for 'growth' of the internal variables. Note that saturation is of course inevitable if there is some low- or high-frequency amplification in the controller, but this can be handled by means of a separate gain factor. The concern here is a more fundamental effect due to the numerical nature of the implementation. An analysis of internal variable overflow for the modified canonic structure has been carried out which implies that the maximum values of $v$, $w$ and $x$ in response to a step input $u=u_{\text{max}}$ are as given below [Forsythe and Goodall (1991)]:
Table 3.1  Intuitive formulation of the maximum values of internal variables in response to a step input $u_{\text{max}}$

<table>
<thead>
<tr>
<th>Internal Variable</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$</td>
<td>$u_{\text{max}}$</td>
</tr>
<tr>
<td>$w$</td>
<td>$&lt; u_{\text{max}}$</td>
</tr>
<tr>
<td>$x$</td>
<td>$2u_{\text{max}}$</td>
</tr>
</tbody>
</table>

The analysis is performed by looking at the initial and final values of all three internal variables in response to a step input and then inferring the maximum values. In the case of $v$ and $x$, the deductions were intuitively straightforward. However, for the case of $w$, simulation was used to illustrate that the maximum value of $w$ was always less than $u_{\text{max}}$ for various damping factors [Forsythe and Goodall (1991)].

An alternative method of determining the maximum value of the internal variables is to consider the $l_\infty$-norm of the internal variable sequences $v(nT)$, $w(nT)$ or $x(nT)$. The $l_\infty$-norm of a vector $v$, supposing $v \in \mathbb{R}^n$ (that is $v$ is a real number), is defined as

$$\|v\|_\infty = \max_{1 \leq j \leq n} |v_j|$$

That is, it is the maximum (absolute) value of $v$ considering all the elements of $v$. This is exactly what one is looking for when analysing the internal variables for overflow requirements.

### 3.2.1.1 $l_\infty$-norm analysis

Consider the internal variable sequence $w(nT)$. The sequence is a result of convoluting the input sequence $u(nT)$ and the impulse response sequence $h_w(nT)$. $h_w(nT)$ is the impulse response sequence between the internal variable $w$ and the input $u$. $w(nT)$ is given by

$$w(nT) = u(nT) \ast h_w(nT)$$

where $\ast$ denotes convolution and can be evaluated as
\[ w(nT) = \sum_{m=-\infty}^{\infty} h_w(mT)u(nT-mT) \]  \hspace{1cm} (3.3)

In equation (3.3), \( w(nT) \) is only defined for \( m > 0 \) and \( w(nT) = 0 \) for \( m \leq 0 \). In order to determine the maximum value of \( w(nT) \), the Hölder inequality can be used by setting \( p = 1 \) and \( q = \infty \) to get [Williamson (1991), pp 534]:

\[ \|w\|_\infty \leq \|h_w\|_1 \|u\|_\infty \]  \hspace{1cm} (3.4)

where \( \|h_w\|_1 \) is the \( l_1 \)-norm of the impulse response sequence \( h_w(nT) \) and is defined as

\[ \|h_w\|_1 = \sum_{j=1}^{n} |h_w(j)| \]  \hspace{1cm} (3.5)

[that is, the \( l_1 \)-norm is the sum of all the absolute values of the impulse response sequence \( h_w(nT) \)].

Equation 3.4 can be interpreted as such: the maximum value of the sequence \( w(nT) \) will be less than or equal to the product of the \( l_1 \)-norm of the impulse response sequence \( h_w(nT) \) and the maximum value of the input sequence \( u(nT) \). If the input sequence \( u(nT) \) is going to be \( u_{\text{max}} \) then (3.4) becomes

\[ \|w\|_\infty \leq \|h_w\|_1 u_{\text{max}} \]  \hspace{1cm} (3.6a)

Similar inequalities can be derived for the internal variables \( v \) and \( x \):

\[ \|v\|_\infty \leq \|h_v\|_1 u_{\text{max}} \]  \hspace{1cm} (3.6b)

\[ \|x\|_\infty \leq \|h_x\|_1 u_{\text{max}} \]  \hspace{1cm} (3.6c)

#### 3.2.1.2 Example using \( l_\infty \)-norm

The above analysis was tested on the problem of implementing a 2nd order low pass 1-Hz filter, transfer function in Appendix A, with various damping factors and a sample frequency of 100 Hz. The simulation results are obtained by using a unit step as an input. The results are as shown in Table 3.2.
Structures for Implementation

<table>
<thead>
<tr>
<th>Damping Factor, $\zeta$</th>
<th>$\nu_{\text{max}}$ Analytical</th>
<th>$\nu_{\text{max}}$ Simulation</th>
<th>$\nu_{\text{max}}$ Analytical</th>
<th>$\nu_{\text{max}}$ Simulation</th>
<th>$\nu_{\text{max}}$ Analytical</th>
<th>$\nu_{\text{max}}$ Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>30.7371</td>
<td>1.0000</td>
<td>2.4595</td>
<td>0.0815</td>
<td>29.713</td>
<td>1.9691</td>
</tr>
<tr>
<td>0.05</td>
<td>13.2967</td>
<td>1.0000</td>
<td>1.9844</td>
<td>0.1509</td>
<td>12.2044</td>
<td>1.8547</td>
</tr>
<tr>
<td>0.5</td>
<td>2.7343</td>
<td>1.0000</td>
<td>1.3882</td>
<td>0.5808</td>
<td>1.3872</td>
<td>1.1633</td>
</tr>
<tr>
<td>0.707</td>
<td>2.4517</td>
<td>1.0000</td>
<td>1.4083</td>
<td>0.6736</td>
<td>1.0886</td>
<td>1.0434</td>
</tr>
</tbody>
</table>

Table 3.2 Analytical and Simulation results of Overflow Analysis

From the above results it is clear that the analytical technique using the $l_\infty$-norm produces results which are pessimistic in practice as would be expected due to the inequality. However, it does provide a analytical alternative to simulation.

From the simulation results in the above table, it can be seen that overflow of the internal variables is not a problem, as generally the internal variables are of a similar size to that of the input; in some cases it may be necessary to allow for an extra bit for overshoot. The numbers resulting from the $l_\infty$-norm method are quite high in comparison to the simulation results. This indicates that were the filter to be driven by an input whose frequency matched the resonant frequency of the filter, then an overflow in the output would occur if an adequate number of bits was not provided. However, for a control system one would have such a filter (e.g. notch) to remove the signals with such unwanted frequencies and hence the overflow situation would not arise. However, in a communication system there might exist situations whereby the filter might be driven by such a signal.

3.2.13 Scaling

Scaling may be interpreted to be a matching of the range of numerical representation with the dynamic range (or range of probable values) of the variables to be represented. It was earlier mentioned that scaling is often performed prior to implementing fixed-point digital filters to reduce the probability of overflow in the internal variables. There are various forms of scaling techniques in use and these

However, with the modified canonic structure, as has been seen above, the very structure means that the internal variables are already scaled and in general, if one neglects overshoot, they do not overflow.

3.2.2 Internal Variable Underflow

Having specified the overflow factor, the situation is as depicted in Figure 3.3. Assuming the coefficient is as shown, one has an integer variable which needs to be multiplied by a coefficient having a certain number of fractional bits. If the product is to be represented to full precision, it will contain as many fractional bits as there are in the coefficient. This product will be the variable the next time and therefore will contain fractional bits. When the variable is now multiplied by a coefficient, it is possible to generate even more fractional bits. The process is recursive, and so eventually precision must be sacrificed; the variable must be truncated or rounded at some precision. The question is - where does one say enough?

A number of methods can be used to analyse the errors that are introduced due to the truncation/rounding of the arithmetical operations. The results of these
Amongst the most popular is the round-off noise analysis. In this method, the quantisation noise introduced due to rounding or truncation is quantified statistically. This method is discussed in section 3.2.2.1. Another way to analyse the situation is to look at it in a deterministic way and perform a steady state analysis. This method is discussed in section 3.2.2.2. Simulation is used to compare and contrast the results of the two methods and this is presented in section 3.2.2.3.

3.2.2.1 Round-off Noise Analysis

In this section the quantisation errors introduced due to rounding/truncation of arithmetic operations will be analysed. Only the results of the multiplication's will be considered since for fixed point computation additions do not generate any round-off noise. In the analysis below only rounding will be considered - as opposed to truncation - for quantisation. This is because of its desirable properties for analysis such as [Rabiner and Gold (1975)]:

- error signal is independent of the type of arithmetic used,
- zero mean unlike truncation and
- no other method of quantisation yields lower variance.

The effects of rounding are modelled by introducing a white noise source in the filter structure whenever a multiplication operation takes place. This degrades the signal to noise ratio in the output. To model the effects of rounding due to quantisation, certain assumptions about the noise sources have to be made [Rabiner and Gold (1975)]. They are:

1/ Any two different samples from the same noise source are uncorrelated.
2/ Any two different noise sources (that is associated with different multipliers) are regarded as random processes and are uncorrelated.
3/ Each noise source is uncorrelated with the input sequence.

Additionally, a number of other assumptions have to be made [Agarwal and Burrus (1975)]. These are:

4/ The input $u(nT)$ is assumed to be a $(b+1)$ bit fraction (one bit is used for sign) with a peak magnitude of 1 and the value of the lowest bit is $2^b$.

5/ All the internal variables also use a $(b+1)$ bits representation such that the least significant bit is always $2^b$.

6/ Rounding / truncation is only necessary if the result exceeds the $(b+1)$ bits allowed.

Therefore, the multiplication operations create round-off noise with a variance $2^{2b}/12$, where $b$ is the number of fractional (underflow) bits of the result.

Consider the modified canonic structure shown in Figure 3.2. For this structure, the result of the $\delta^1$ operations (basically additions) does not need to be rounded. Hence, there is no need to introduce a noise source. Similarly, the calculation of $v = u - w - x$ does not introduce a noise source. For the above structure, only the following noise sources are introduced:

- $e_1(nT)$, due to the multiplication between $d_i$ and $v_i$ is assumed to be white noise between $\pm 2^{b+1}$ with variance $\sigma_{e_i}^2 = 2^{2b}/12$
- $e_2(nT)$, due to the multiplication between $d_j$ and $w_j$ is assumed to be white noise between $\pm 2^{b+1}$ with variance $\sigma_{e_j}^2 = 2^{2b}/12$
- $e_3(nT)$, due to the arithmetic operation $(pv + qw + rx)$, which is the sum of three independent noise sources lying between $\pm 2^{b+1}$ with the variance of each being $2^{2b}/12$. Therefore, $\sigma_{e_3}^2 = 3 \times 2^{2b}/12$

Hence the structure shown in Figure 3.4 can be deduced.
For analysis, the output noise power due to these arithmetical noise sources is of interest. Therefore, consider the following: for a random white noise input, having variance \( \sigma_i^2 \), passing through a transfer function \( H(z) \), the output noise power \( \sigma_o^2 \) is given by [Agarwal and Burrus (1975)]:

\[
\sigma_o^2 = \sigma_i^2 H_r^2
\]

(3.7)

where \( H_r^2 \) is defined as the r.m.s transfer function magnitude

\[
H_r^2 = \frac{T}{2\pi} \int_{-\pi}^{\pi} |H(e^{i\omega T})|^2 d\omega
\]

(3.8)

It is difficult to perform the integration to solve for \( H_r^2 \) in closed forms and numerical integration has to be performed to obtain a result. Alternatively, equation (3.8) can also be evaluated by calculating the following summation which is arrived at by making use of Parseval’s Relation [Rabiner and Gold (1975), pp 36]

\[
\frac{T}{2\pi} \int_{-\pi}^{\pi} H(e^{i\omega T})^2 d\omega = \sum_{n=-\infty}^{\infty} h^2(nT)
\]

(3.9)

Again, it is quite difficult to perform the desired summation and the computations can be eased by evaluating the infinite sum as
where $H(z)$ is the z-transform of the impulse response sequence $h(nT)$ and is readily obtained.

Therefore, one needs to calculate the transfer functions from the various noise sources to the output and find the r.m.s transfer function magnitude for each by solving the integral in equation (3.10) for each transfer function.

**Evaluating r.m.s transfer function magnitudes**

The contour integral can be evaluated by a number of different methods as outlined in Appendix B. The solutions for $H_1, H_2, H_3$ are given in Appendix B. Of all the methods the easiest to handle was the simple method outlined in [Mitra, et al (1974)]. Thus, this method will be used from now to calculate the r.m.s transfer function magnitudes.

However, of the methods outlined in Appendix B, consider briefly the emulation method that is used to calculate the r.m.s transfer function magnitudes. To the author's knowledge this method is believed to be a novel, but not necessarily easier, way of solving for the r.m.s transfer function magnitudes. Since the method has not been used before it is worth examining the results from it.

At first sight it may seem that the results from the emulation method are not very accurate. This is clear from columns 3 and 5 in Table B.1. This is to be expected since the emulation is only an approximation. However, it should be noted that as the sampling frequency, $f_s$, is increased with respect to the dominant frequency of the controller, the result from the emulation method will approach the result of the contour integration. Consider the implementation of a 1 Hz low pass filter, damping factor 0.707, with the sample frequency increased progressively from 10 Hz to 1000 Hz. The r.m.s transfer function magnitudes $H_{1r}$ and $H_{2r}$ calculated using Mitra et al's simple method and using the emulation method are shown in Table 3.3.
Table 3.3 Variation of results from Mitra et al.'s and emulation method as sample frequency is increased

<table>
<thead>
<tr>
<th></th>
<th>$f_i = 10$ Hz</th>
<th>$f_i = 100$ Hz</th>
<th>$f_i = 1000$ Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$H_{1R}^2$</td>
<td>$H_{2R}^2$</td>
<td>$H_{1R}^2$</td>
</tr>
<tr>
<td>Mitra et al</td>
<td>0.29474</td>
<td>1.688</td>
<td>2.8172</td>
</tr>
<tr>
<td>Emulation</td>
<td>0.31905</td>
<td>1.8697</td>
<td>2.82</td>
</tr>
</tbody>
</table>

It is clear from the table that as the sampling frequency, $f_i$, is increased, the results of the emulation method approach those obtained using Cauchy's residue theorem.

Having calculated the r.m.s transfer function magnitudes, the next stage is to calculate the variance of the output signal due to the noise sources. Using assumption 2 given by [Rabiner and Gold (1975)]:

$$\sigma_e^2 = H_{1R}^2 \sigma_{e_1}^2 + H_{2R}^2 \sigma_{e_2}^2 + H_{3R}^2 \sigma_{e_3}^2$$  \hspace{1cm} (3.11)

where $H_{kr}^2$ ($k = 1, 2, 3$) is the r.m.s transfer function magnitude between the kth noise source and the output $y$ - evaluated using equations (3.8) to (3.10) above; $\sigma_{e_1}^2$ ($=2^{2b}/12$), $\sigma_{e_2}^2$ ($=2^{2b}/12$) and $\sigma_{e_3}^2$ ($=3.2^{2b}/12$) are the variances of the noise sources $e_1$, $e_2$ and $e_3$, respectively. Substituting these values into (3.11) and rearranging, gives

$$b = \frac{1}{2} \log_2 \left( \frac{H_{1R}^2 + H_{2R}^2 + 3H_{3R}^2}{12\sigma_e^2} \right)$$  \hspace{1cm} (3.12)

Equation (3.12) expresses the number of fractional bits $b$ as a function of the variance of the output signal. Therefore, it should be possible to calculate the number of bits required for the internal variables to obtain an output signal with a certain variance.

**Distribution of Output Samples**

In order to use equation (3.12) to calculate the number of fractional bits, $b$, it is important to understand that the samples of the output signal will be normally distributed. Consider how the output sequence is formed: the output sequence $y(nT)$ is a sum of three sub-output sequences, $y_1(nT)$, $y_2(nT)$ and $y_3(nT)$ (each from
Now, consider the case of how \( y_i(nT) \) is formed. \( y_i(nT) \) is the result of a convolution between the impulse response \( h_i(nT) \) and \( e_i(nT) \). Now, the strength of each of the samples of \( e_i(nT) \) is taken from a uniform distribution and, at each instant in time, each of these samples can be thought of as an impulse to the system.

It is well known that an impulse, of strength, say \( I_i \), will produce a certain output sequence, say \( Y_i \). The samples of the output sequence will always have a certain shape over time. Now an impulse, \( I_j \) of a separate strength, say \( 2I_i \), will produce a output sequence, say \( Y_j \). The samples of \( Y_j \) will have the same shape as those from \( Y_i \) but the amplitude of the samples in \( Y_j \) will be twice that of each corresponding sample in \( Y_i \). It is therefore obvious that each sample in an output sequences will be part of a uniform distribution. When \( y_i(nT) \) is formed, what happens is that all the samples are shifted as appropriate and added together.

That is, samples which are from similar distributions are added together and, therefore, by the central limit theorem, all the resulting samples will be part of a normal distribution. The final output \( y(nT) \) is a sum of samples which are from a normal distribution and therefore the resultant samples will also be part of a normal distribution.

With this understanding equation (3.12) can be used with comfort to calculate for \( b \) by ensuring that the variance of the output signal is no more than a certain variance.
structures for implementation. This will be seen in section 3.2.2.3.

3.2.2.2 Steady State Analysis

In this section the underflow issue is analysed by posing the problem in a deterministic way. As was seen the underflow bits provide the internal variables with fractional bits. With the input and all internal variables initially set to zero, nothing will happen. If the input changes from 0 to 1 (that is, the smallest possible change, given the position of the binary point as depicted in Figure 3.1), the output must also change in a manner appropriate to the particular controller being implemented. If the output does not respond, then the original specification for the resolution of the input variable has been contravened. It is intuitively obvious that the problem relates to the number of fractional bits which are provided. By posing the question, 'will it respond appropriately, or will it not?', the problem becomes deterministic [Forsythe and Goodall (1991), pp 143-4].

A steady state analysis for the underflow bits needed for the modified canonic structure has been carried out in [Forsythe and Goodall (1991), pp 163-5]; therefore, it will not be repeated here. The underflow (or fractional) bits assuming the results of the multiplication's are rounded can be evaluated using

\[ b = \log_2 \left( \frac{d_1 + d_2}{d_1 d_2 e_{q}} \right) - 1 \]  

(3.13)

where \( e_{q} \) is the maximum fractional error which can be tolerated in the value of internal variable \( x \) after it is quantised.

3.2.2.3 Comparison of Analyses

Having performed the analysis, one can now determine the effectiveness of the two methods. To do this consider the problem of implementing a 10 Hz notch filter with a damping factor of 0.05, and a 10 Hz low pass filter with a damping factor of 0.707 (both transfer functions are given in Appendix A). Both filters are examined
for sampling frequencies of 100 Hz, 500 Hz, 1000 Hz and 10 kHz. Also, suppose the aim is to achieve no more than a 5% error in the output.

Using equation (3.13) and setting $\epsilon_n$ to be equal to 0.05, the number of underflow bits for the steady state analysis can be easily calculated.

For round-off noise analysis, the problem is of relating the maximum fractional error to the variance of the output signal. It is known that the samples of the output are from a normal distribution. For a normal distribution it is also known that 95% of the output samples will be within $\pm 2\sigma$. So it can be said that the maximum value of the output (or error) due to the round-off noise sources should be 0.05 and work to say a 95% confidence limit. That is,

$$2\sigma = 0.05 \Rightarrow \sigma^2 = (0.05/2)^2 = 0.00625 \times 10^4.$$

Setting the above value for $\sigma^2$ in equation (3.12), the number of underflow bits using round-off noise analysis can also be calculated.

Predictions of the number of underflow bits using the two methods are given in Table 3.4, based upon achieving no more than a 5% error in the output.

<table>
<thead>
<tr>
<th>Sample Frequency (Hz)</th>
<th>Low pass Filter</th>
<th>Notch Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Round-off</td>
<td>Steady State</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>500</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>10000</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3.4 Underflow bits as predicted by round-off noise and steady state analysis for the modified canonic structure

From Table 3.4 it is seen that there is great disagreement between the round-off noise analysis and steady state analysis. Additionally, the difference between the two increases as the sample frequency is increased with respect to the filter frequency. The important question is which of the two is correct and what effect does the difference have on the output? Simulation is therefore used to assess the
The response of the filters to a step input (effectively a change in the least significant bit of the input) is examined. Figures 3.6 and 3.8 show the results for the low pass filter for sample frequencies of 500 Hz and 10 kHz respectively and Figures 3.9 and 3.11 show the results for the notch filter, again for sample frequencies of 500 Hz and 10 kHz.

From Figure 3.6 it is not clear whether the error in the output due to the underflow bits predicted by the round-off noise analysis is within the bounds that are being worked to, that is \( \leq 5\%\). The steady state result is however less than 5\%. As a result, a graph of the error in the output of the low pass filter sampled at 500 Hz is given in Figure 3.7. This shows that the error from both is within the allowable 5\% limit. Therefore, the results of the round-off noise analysis could easily be used and therefore 6 bits would be used in practice.

![Graph showing step response of a 10 Hz low pass filter sampled at 500Hz](image)

*Figure 3.6 - Step response of a 10 Hz low pass filter sampled at 500Hz*
However, at higher sample frequencies, in this case 10 kHz, Figure 3.8 clearly shows that the underflow bits predicted by the round-off noise analysis are not enough as the error in the output, about 20%, far exceeds the maximum allowable value of 5%. So this time the underflow bits predicted by the round-off noise analysis are not enough and the steady state analysis results gives a safe or conservative prediction.
Figure 3.8 - Step response of a 10 Hz low pass filter sampled at 10 kHz

For the notch filter sampled at 500 Hz, Figure 3.9 shows that the number of bits predicted by both analyses is sufficient and a quick check of the error plot, shown in Figure 3.10, confirms this. Therefore, it could be argued that for this case the predictions of the round-off noise analysis are good enough.
Figure 3.9 - Step response of a 10 Hz notch filter sampled at 500 Hz

Figure 3.10 - Error in output of a 10 Hz notch filter sampled at 500 Hz
But, what happens when the sample frequency is increased? The result for a sample frequency of 10 kHz is shown in Figure 3.11. In this instance the number of bits predicted by round-off noise analysis are not enough while the number predicted by steady state analysis are more than (!) enough. In this particular case, the result of the round-off noise prediction, unlike the result of the steady state prediction, is not following the exact response; the demand is a minimum change in the input. A plot of the error in the output for this case is shown in Figure 3.12. Figure 3.12 shows that the error in the round-off noise analysis decreases with time and eventually this error does fall below the maximum allowable limit because the output is decaying with time and not because the output of the round-off noise prediction is following the exact response!

To summarise, in general the round-off noise analysis is overly optimistic and does not always predict a sufficient wordlength for the internal variables, especially at high sample rates. On the other hand, the steady state analysis is the more reliable method, although it tends to over-specify the wordlength.

![Figure 3.11 - Step response of a 10 Hz notch filter sampled at 10 kHz](image-url)
The Cross Coupled Modified Canonic Structure

The wordlength requirements for a first order modified canonic section are less than those for a second order modified canonic section. So the question whether one could implement the second order section as some combination of two first order sections arises. If there is a solution, then the wordlength advantages of the first order sections could be exploited while still implementing a second order function. The solution is trivial if the roots of the second order section are real and not repeated; the second order section is then easily implemented as a parallel or cascade combination of two independent first order sections. However, what if the second order section has complex or a pair of real repeated roots? Is some combination of first order sections still possible?
A combination of two first order structures in parallel with cross coupling between the two can be created. This is achieved by feeding the internal variable, \( v \), of each of the two first order sections via a gain to the other first order section; hence, the cross coupling. This is depicted in Figure 3.13.

\[
\begin{align*}
\frac{y}{u}(s) &= \frac{p_c + q_c s^2 + r_c s^3}{1 + d_{c1} s^3 + d_{c2} s^4} \\
\end{align*}
\] (3.14)

where

\[
\begin{align*}
p_c &= p_1 + p_2 \\
q_c &= q_1 (d_{12} - d_{21}) + p_2 (d_{11} - d_{21}) + q_1 (d_{11} + d_{12}) + q_2 (d_{12} + d_{22}) \\
r_c &= (q_2 d_{11} + q_1 d_{12}) (d_{22} - d_{12}) + (q_2 d_{12} + q_1 d_{22}) (d_{11} - d_{21}) \\
d_{c1} &= d_{11} + d_{22} \\
d_{c2} &= d_{11} d_{22} - d_{21} d_{12}
\end{align*}
\]
The corresponding equations to implement the cross coupled structure are obvious, and transform to:

\[ v = u - w \]
\[ y = pv +qw \]

and the \( \delta^i \) operations are calculated as

\[ w = w + d^i v \]

where

\[
\begin{align*}
    u &= \begin{bmatrix} f \\ f \end{bmatrix}, \\
    v &= \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \\
    w &= \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \\
    p &= \begin{bmatrix} p_1 & p_2 \end{bmatrix}, \\
    q &= \begin{bmatrix} q_1 & q_2 \end{bmatrix}, \\
    d^i &= \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix}
\end{align*}
\]

### 33.1 Calculating the coefficient values

The cross coupled structure can be used to represent a second order modified canonic structure whose transfer function is given by

\[
\frac{y}{u}(\delta) = \frac{p + q d_1 \delta^{-1} + r d_1 d_2 \delta^{-2}}{1 + d_1 \delta^{-1} + d_1 d_2 \delta^{-2}}
\]

Comparing equations (3.14) and (3.15), it can be seen that

\[
p_c = p, \quad q_c = qd_1, \quad r_c = rd_1, \quad d_c = d_1, \quad d_x = d_1 d_2
\]

Since the values \( p, q, r, d_1 \) and \( d_2 \) are known, the values \( p_c, q_c, r_c, d_c \) and \( d_x \) can be easily calculated. Now there is a problem: there are five linear equations and eight unknowns \( (p, q, r, d_1, d_2, d_3, d_4, d_5) \). Thus, some assumptions about the unknown coefficients need to be made to reduce the number of unknowns.

The first assumption is made regarding the forward path coefficients \( p \) and \( p_c \). A number of different simplifications are possible but the one that worked for a number of different controllers is the case where \( p_c = p = p_c/2 \). This is computationally convenient because only one multiplication is needed if \( v_1 \) and \( v_2 \).
Structures for Implementation

are added first. This reduces the number of unknowns by one to seven.

The second assumption is made depending on the poles of the controller being implemented. The poles (or roots of the denominator) are either going to be complex conjugate pair or a pair of real repeated roots, and are a function of $d_{1\nu}, d_{2\nu}, d_{1r}$ and $d_{2r}$. For both cases the Jordan form of representing a matrix can be used:

Recall that a matrix, say $A$, can be transformed to the Jordan form, $A_J$, by means of similarity transform. The resulting matrix $A_J$ can have three different forms depending on the eigenvalues of $A$.

If all the eigenvalues of $A$ are real non-repeated then $A_J$ is a diagonal matrix with the diagonal elements being the eigenvalues of $A$. All the other off-diagonal terms are zero. That is,

$$
A_J = \begin{bmatrix} 
\sigma_1 & 0 & \cdots & 0 \\
0 & \sigma_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & \sigma_s 
\end{bmatrix} 
$$

(3.16a)

If $A$ has some complex eigenvalues, then each complex conjugate pair, $\sigma_k \pm j\omega_k$, appears in $A_J$ as a 2x2 block with $\sigma_k$ as the diagonal terms and $\omega_k$ as the off-diagonal terms as shown in equation (3.16b). Any real eigenvalues appear as in equation (3.16a).

$$
A_J = \begin{bmatrix} 
\sigma_k & \omega_k \\
-\omega_k & \sigma_k 
\end{bmatrix} 
$$

(3.16b)

Finally, if $A$ has real repeated eigenvalues, then this appears as a diagonal block as given in equation (3.16c).

$$
A_J = \begin{bmatrix} 
\sigma_i & 0 & 0 & \cdots & 0 \\
1 & \sigma_i & 0 & \cdots & 0 \\
0 & 1 & \sigma_i & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & \sigma_i 
\end{bmatrix} 
$$

(3.16c)
So depending on the poles of the controller, the Jordan form representations in equations (3.16b) and (3.16c) can be applied to the $d_4$ matrix.

33.1.1 Complex roots

The Jordan form for complex conjugate pair, given in equation (3.16b), can be used as basis for the choice of the $d_4$ matrix. That is,

\[
\begin{bmatrix}
\sigma & \omega \\
-\omega & \sigma
\end{bmatrix}
\Leftrightarrow
\begin{bmatrix}
d_{11} & d_{12} \\
d_{21} & d_{22}
\end{bmatrix}
\]

that is

\[
\text{let } d_{11} = d_{22} \quad d_{21} = -d_{12}
\]

After making the above two assumptions, the number of unknowns have now been reduced to five and the five linear equations that are available can be used to solve for them.

Simulations were carried out using the above approach for the following four filters, each of which has complex poles. The transfer function for each is given in Appendix A.

- A low pass filter with a corner frequency, $f_c$, of 10 Hz, damping factor, $\zeta$, of 0.707. This filter was emulated to the discrete form using the bilinear transform at a sample frequency, $f_s$, of 1000 Hz. The response of the filter to a step input was examined and the results for the cross coupled system and the continuous system were similar. Similarly, simulations were carried out on a range of different transfer functions (ie varying $\zeta$ and $f_c$ independently) and in each case the results of the cross coupled and the continuous systems were in good agreement.

The effect of different sampling rates, for a fixed transfer function, was also examined and once again the responses of the cross coupled and continuous systems were as expected. That is, at high sample rates (as
compared to \( f_c \) the responses were almost identical but at lower sample rates the errors due to emulation were obvious.

Therefore, only the case of \( f_c = 10 \text{ Hz}, \zeta = 0.707 \) and \( f_s = 100 \text{ Hz} \) and \( f_s = 1 \text{ kHz} \) is shown in Figure 3.14. The response of the cross coupled system at \( f_s = 100 \) Hz is similar to the continuous system and as the sample frequency is increased, the difference in response is greatly reduced. The difference is essentially due to emulation of the continuous system.

![Figure 3.14 - Comparing step responses of continuous and discrete (cross coupled implementation) forms of 10 Hz low pass filter sampled at 100 Hz and 1 kHz](image)

- The second filter simulated is a band reject (or notch) filter. Once again the continuous system is emulated to the discrete equivalent using the bilinear transform and the response of the systems to a step input is studied. Once again the results of the two systems are identical and the case for \( f_c = 10 \text{ Hz}, \zeta = 0.05, f_s = 100 \text{ Hz} \) and \( f_s = 1 \text{ kHz} \) is shown in Figure 3.15.
The third type of filter simulated is a high pass filter. The procedure followed is identical to the one described above and the responses to a step input for both the continuous and discrete systems are similar.

The fourth type of filter is the band pass filter and responses of both continuous and discrete systems to a step input were examined for a range of $\zeta$'s, $f_c$'s and $f_s$'s. The results of the two systems are similar, allowing for emulation error.

33.1.2 Repeated roots (one pair only)

For this case a second order block from equation (3.16c) can be imposed upon the $d_1$ matrix, that is
Structures for Implementation

\[
\begin{bmatrix}
\sigma & 0 \\
1 & \sigma
\end{bmatrix} \Leftrightarrow \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix}
\]

that is

let \( d_{11} = d_{22} \quad d_{21} = 1 \quad d_{12} = 0 \)

The above assumptions can be tested out on the filters given in section 3.3.1.1 with \( \zeta = 1, f_c = 1/2\pi \text{ Hz} \); this gives two roots at \( s = -1 \). The results of the continuous and discrete systems are similar and are shown in Figure 3.16 for a low pass filter with the above values of \( \zeta \) and \( f_c \), and for sample frequencies of \( f_s = 10/2\pi \text{ Hz} \) (10 times \( f_c \)) and \( f_s = 100/2\pi \text{ Hz} \) (100 times \( f_c \)).

![Figure 3.16 - Comparing step responses of continuous and discrete form (cross coupled implementation) of a low pass filter with real repeated roots](image)

To determine the number of overflow bits that need to be provided, the maximum values of the internal variables, \( v_r, v_r, w_t, \) and \( w_r \), in response to a maximum input \( u_{\text{max}} \) have to be evaluated. Two methods can be used: an initial and final value analysis.
of the variables to a step input \( u = u_{\text{max}} \) can be carried out by substituting \( \delta^1 = 0 \) and \( \infty \) respectively and then the maximum values are then deduced intuitively; this analysis is similar to the one performed by [Forsythe and Goodall (1991)]; another way of determining the maximum value is to perform an \( L_\infty \) norm analysis, similar to the one described in section 3.2.1.2 for the modified canonic structure.

### 3.3.2.1 Initial and Final value analysis

This method involves analysing discrete transfer functions in \( \delta \). The procedure is algebraically tedious and can be error prone; for further details of the transfer functions see Appendix D. The initial and final values to a step input \( u = u_{\text{max}} \) can be determined by substituting \( \delta^1 = 0 \) and \( \infty \) respectively in the transfer functions; this yields the values shown in Table 3.5 for the cross coupled structure.

<table>
<thead>
<tr>
<th>Internal Variable</th>
<th>Initial (( \delta^1 = 0 ))</th>
<th>Final (( \delta^1 = \infty ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>( u_{\text{max}} )</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>( u_{\text{max}} )</td>
<td>0</td>
</tr>
<tr>
<td>( w_1 )</td>
<td>0</td>
<td>( u_{\text{max}} )</td>
</tr>
<tr>
<td>( w_2 )</td>
<td>0</td>
<td>( u_{\text{max}} )</td>
</tr>
</tbody>
</table>

*Table 3.5 Initial and final value analysis of internal variables in response to input \( u_{\text{max}} \) for the cross coupled structure*

From the table it can be seen that \( v_1 \) and \( v_2 \) both start at \( u_{\text{max}} \) and then settle to zero; therefore, one can infer that there may be some undershoot to a negative value before settling, but that the maximum value is likely to be the initial value, that is \( u_{\text{max}} \). Variables \( w_1 \) and \( w_2 \) start at zero and settle to \( u_{\text{max}} \) possibly with overshoot to nearly twice this value for low-damped filters.

Since the above deductions are only intuitive, simulation is performed to check whether the above deductions are correct. The following filters were simulated to determine the response of the internal variables to a maximum input \( u_{\text{max}} = 1 \); their
Structures for Implementation

transfer functions in the s-domain are given in Appendix A.

- Low pass filter
- Band reject (notch) filter
- High pass filter
- Band pass filter

Each filter was simulated for a range of damping factors and the internal variables analysed. The results from each of the filters leads to identical conclusions for all the internal variables. That is the internal variables have identical properties as far as overflow is concerned for each of the filters.

**Internal variable \(v\)**
The initial value of \(v\) is always \(u_{\text{sat}}\) and this immediately starts decaying to zero. The maximum value of \(v\) is \(u_{\text{sat}}\) for filters which are highly damped and just greater than \(u_{\text{sat}}\) when the filter has a low damping factor (eg. \(\zeta \leq 0.05\)). Figure 3.17 shows various curves of how the amplitude of \(v\) varies with time in a notch filter, \(f_c = 1 \text{ Hz}\) and \(f_s = 100 \text{ Hz}\), for a range of \(\zeta\)'s.

**Internal variable \(v\)**
\(v\) also has an initial value of \(u_{\text{sat}}\) and also decays to zero. However, for \(v\), the maximum value is greater than \(u_{\text{sat}}\) for \(\zeta \leq 0.5\) but the maximum value does not approach \(2u_{\text{sat}}\). Figure 3.18 shows various curves of how the amplitude of \(v\) varies with time in a notch filter, \(f_c = 1 \text{ Hz}\) and \(f_s = 100 \text{ Hz}\), for a range of \(\zeta\)'s.

**Internal variable \(w\)**
\(w\) has an initial value of zero and this eventually settles to \(u_{\text{sat}}\). The maximum value is about \(2u_{\text{sat}}\) for \(\zeta \leq 0.1\) and between \(u_{\text{sat}}\) and \(2u_{\text{sat}}\) for \(0.707 \leq \zeta \leq 0.1\). Figure 3.19 shows various curves of how the amplitude of \(w\) varies with time in a notch filter, \(f_c = 1 \text{ Hz}\) and \(f_s = 100 \text{ Hz}\), for a range of \(\zeta\)'s.
Internal variable $w$.

The initial and final values of $w$, zero and $u_{\text{max}}$ respectively. The maximum value is as for $w$. Figure 3.20 shows various curves of how the amplitude of $w$ varies with time in a notch filter, $f_\ell = 1$ Hz and $f_s = 100$ Hz, for a range of $\zeta$'s.

Figure 3.17 - Simulation to show size of $v_1$ in response to $u = 1$
Figure 3.18 - Simulation to show size of $v_1$ in response to $u = 1$

Figure 3.19 - Simulation to show size of $w_1$ in response to $u = 1$
From the above simulation studies it can therefore be concluded that at most one needs to provide one overflow bit for the internal variables. In most cases this is when the filter has a low damping factor. This bit will account for a 100% overshoot in the internal variables.

### L₂-norm analysis

By performing an analysis similar to the one performed for the modified canonic structure in section 3.2.1.1, the following inequalities can be deduced for the internal variable sequences \(v_1(nT), v_2(nT), w_1(nT)\) and \(w_2(nT)\).

\[
\begin{align*}
\|v_1\|_\infty & \leq \|h_{v_1}\|_1 u_{\text{max}} & (3.17) \\
\|v_2\|_\infty & \leq \|h_{v_2}\|_1 u_{\text{max}} & (3.18) \\
\|w_1\|_\infty & \leq \|h_{w_1}\|_1 u_{\text{max}} & (3.19) \\
\|w_2\|_\infty & \leq \|h_{w_2}\|_1 u_{\text{max}} & (3.20)
\end{align*}
\]
Structures for Implementation

The results of the above analysis are similar to those obtained for the modified canonic structure. That is, the results of the analytical technique are correct but not very practical to use. They are therefore not displayed here.

Once again from simulation it can be seen that the overflow of the internal variables is not a problem in the structure as, generally, the internal variables are of a similar size to that of the input; additionally note that in some cases it may be necessary to allow for an extra bit for overshoot.

Internal Variable Underflow

Having considered the overflow of the internal variables, once again the situation is as depicted in Figure 3.3. The argument for providing a number of underflow bits as described in section 3.2.2 still holds. Therefore, the methods which can be used to predict the number of underflow bits that are needed to achieve a required percentage accuracy in the output needs to be looked at. As for the modified canonic structure, the round-off noise analysis, described in section 3.3.3.1, and steady state analysis, described in section 3.3.3.2, are used. The results of the two analyses are compared in section 3.3.3.3 using simulation.

Round-off Noise Analysis

The assumptions made in section 3.2.2.1 for the modified canonic structure still hold for the cross coupled structure. Thus, for the cross coupled structure shown in Figure 3.13, the following noise sources are introduced:

- \( e_{nT} \), which is a sum of two independent noise sources lying between \( \pm 2^{b-1} \) with the variance of each being \( 2^{-b}/12 \). The two noise sources are due to the multiplication between \( d_{n} \) and \( v_{i} \) and due to the multiplication between \( d_{n} \) and \( v_{i} \). Therefore, \( \sigma_{e_{nT}}^2 = 2.2^{-b}/12 \).
Structures for Implementation

- \( e(nT) \), which is a sum of two independent noise sources lying between \( \pm 2^{b+1} \) with the variance of each being \( 2^b/12 \). The two noise sources are due to the multiplication between \( d_{i1} \) and \( v_i \) and due to the multiplication between \( d_{i2} \) and \( v_i \). Therefore, \( \sigma_{e_i}^2 = 2.2^b/12 \).

- \( e(nT) \), due to the arithmetic operation \( (p_i v_i + q_i w_i + p_i w_i + q_i v_i) \), which is the sum of four independent noise sources lying between \( \pm 2^{b+1} \) with the variance of each being \( 2^b/12 \). Therefore, \( \sigma_{e_i}^2 = 4.2^b/12 \).

Hence the structure shown in Figure 3.21 can be deduced.

![Figure 3.21 - Cross coupled structure with noise sources](image)

The variance of the output signal for the cross coupled structure is given by:

\[
\sigma_o^2 = H_{iR}^2 \sigma_{e_i}^2 + H_{iR}^2 \sigma_{e_i}^2 + H_{iR}^2 \sigma_{e_i}^2
\]

(3.21)

where, as before, \( H_{iR}^2 \) (\( k = 1, 2, 3 \)) is the r.m.s transfer function magnitude between the \( k \)th noise source and the output \( y \) and is evaluated as outlined in Appendix B; \( \sigma_{e_i}^2 (=2.2^b/12) \), \( \sigma_{e_i}^2 (=2.2^b/12) \) and \( \sigma_{e_i}^2 (=4.2^b/12) \) are the variances of the noise sources \( e_i \), \( e_i \) and \( e_i \), respectively. Substituting these values into (3.21) and...
rearranging gives

\[ b = \frac{1}{2} \log_2 \left( \frac{2H_1^2 + 2H_3^2 + 4H_5^2}{12 \sigma_0^2} \right) \]  (3.22)

Equation (3.22) expresses the number of fractional bits \( b \) as a function of the variance of the output signal as before. Hence, it should be possible to calculate the number of bits required for the internal variables to obtain an output signal with a certain variance.

33.2 Steady State Analysis

Steady state analysis examines the underflow issue in a deterministic manner. The analysis derives expressions for the fractional error due to quantisation both in the output and in the internal variables. First consider the equations which are used to perform the necessary calculations in a cross coupled structure. These are:

\[
\begin{align*}
v_i &= u - w_i \\
v_2 &= u - w_2 \\
w_i &= w_i + d_1 v_i + d_2 v_2 \\
w_2 &= w_2 + d_1 v_2 + d_2 v_2 \\
y &= p_i v_i + q_i w_i + p_2 v_2 + q_2 w_2 
\end{align*}
\]

A steady state following a step input is only achieved when \( w_i \) and \( w_2 \) no longer change, that is when the results of \( (d_1 v_i + d_2 v_2) \) and \( (d_1 v_2 + d_2 v_2) \) are both zero. Let \( v_{\text{sq}} \) be the quantised value of \( v_i \) at steady state and so on for \( v_2, w_i, \) and \( w_2 \) and assume that the results of the multiplications are truncated, the steady state is when

\[
\left\lfloor \frac{d_{11} v_{\text{sq}} 2^b}{2^b} \right\rfloor + \left\lfloor \frac{d_{12} v_{\text{sq}} 2^b}{2^b} \right\rfloor = 0 
\]  (3.23)

and

\[
\left\lfloor \frac{d_{21} v_{\text{sq}} 2^b}{2^b} \right\rfloor + \left\lfloor \frac{d_{22} v_{\text{sq}} 2^b}{2^b} \right\rfloor = 0 
\]  (3.24)

In the above equations, \( \lfloor x \rfloor \) represents the floor function, that is round down to the nearest integer towards \( -\infty \), and \( b \) is the number of fractional bits. The next stage is to consider the conditions under which equations (3.23) and (3.24) are satisfied.
Begin by considering equation (3.23); this can also be written as

\[ L d_{21} v_{1s} 2^b + L d_{22} v_{2s} 2^b = 0 \]

\[ \Rightarrow A + B = 0 \]

This can be satisfied by either one of the following conditions

\[ A = 0 \text{ and } B = 0 \]

\[ \text{or} \]

\[ A = -1 \text{ and } B = 1 \]

\[ \text{or} \]

\[ A = -2 \text{ and } B = 2 \]

that is, when the following inequalities are met:

\[ 0 \leq d_{11} v_{1s} 2^b < 1 \text{ and } 0 \leq d_{12} v_{2s} 2^b < 1 \]

\[ \text{or when} \]

\[ -1 \leq d_{11} v_{1s} 2^b < 0 \text{ and } 1 \leq d_{12} v_{2s} 2^b < 2 \]

\[ \text{or when} \]

\[ -2 \leq d_{11} v_{1s} 2^b < -1 \text{ and } 2 \leq d_{12} v_{2s} 2^b < 3 \]

and so on.

Before considering equation (3.24), an assumption regarding the coefficients \( d_{11} \) and \( d_{12} \) needs to be made. If it is assumed that the controller being implemented has complex poles, then \( d_{12} = -d_{11} \) and \( d_{12} = d_{11} \). By making these substitutions, equation (3.24) gives the following inequalities

\[ 0 \leq d_{11} v_{1s} 2^b < 1 \text{ and } 0 \leq d_{12} v_{2s} 2^b < 1 \]

\[ \text{or when} \]

\[ 1 \leq d_{11} v_{1s} 2^b < 2 \text{ and } 1 \leq d_{12} v_{2s} 2^b < 2 \]

\[ \text{or when} \]

\[ 2 \leq d_{11} v_{1s} 2^b < 3 \text{ and } 2 \leq d_{12} v_{2s} 2^b < 3 \]

and so on.

The relationships which satisfy equations (3.23) and (3.24) are depicted in Figure 3.22. The graph has been drawn assuming \( d_{11} > d_{12} \), that is \( 1/d_{11} < 1/d_{12} \).
From the graph it can be seen that both equations (3.23) and (3.24) are satisfied when the inequalities for the two intersect. The boundary or upper limit for the intersection is $1/d_{11}$ for $v_{1_{\text{seq}}}$ and $v_{2_{\text{seq}}}$. If, however, the graph were to be drawn assuming $d_{12} > d_{11}$, then the upper limit of the intersection would be given by $1/d_{12}$ for both $v_{1_{\text{seq}}}$ and $v_{2_{\text{seq}}}$. Therefore, in conclusion, the larger of $[d_{11}, d_{12}]$ defines the maximum value that $v_{1_{\text{seq}}}$ and $v_{2_{\text{seq}}}$ can take to satisfy the steady state conditions. That is,
Structures for Implementation

If \( d_{ll} > d_{12} \) then \( \hat{v}_{1sq} = \frac{2^b}{d_{ll}} \) and \( \hat{v}_{2sq} = \frac{2^b}{d_{ll}} \)

and if \( d_{12} > d_{ll} \) then \( \hat{v}_{1sq} = \frac{2^b}{d_{12}} \) and \( \hat{v}_{2sq} = \frac{2^b}{d_{12}} \)

and if \( d_{ll} = d_{12} \) then \( \hat{v}_{1sq} = \frac{2^b}{d_{ll}} \left( = \frac{2^b}{d_{12}} \right) \) and \( \hat{v}_{2sq} = \frac{2^b}{d_{ll}} \left( = \frac{2^b}{d_{12}} \right) \)

Therefore, assuming \( d_{ll} > d_{12} \), the following inequalities can be deduced for \( v^\prime\prime\prime\prime \) and \( v^\prime\prime\prime\prime\prime \) to obtain a steady state condition.

\[
0 \leq v_{1sq} < \frac{2^b}{d_{ll}} \quad (3.25)
\]

\[
0 \leq v_{2sq} < \frac{2^b}{d_{ll}} \quad (3.26)
\]

The expressions in equations (3.25) and (3.26) indicate the maximum errors in \( v \), and \( v' \). The maximum error in \( w_i \) and \( w_j \) is derived as follows. It is possible to write down an expression for the quantised steady state value \( w_{1sq} \) and \( w_{2sq} \) in response to a minimum input \( u = 1 \) as:

\[
w_{1sq} = 1 - v_{1sq}
\]

\[
w_{2sq} = 1 - v_{2sq}
\]

Substituting the range of errors for \( v_{1sq} \) and \( v_{2sq} \) gives:

\[
l \geq w_{1sq} > 1 - \frac{2^b}{d_{ll}}
\]

In section 3.3.2.1 when a simulation of the internal variables was performed, it was seen that at steady state \( w_i = w_j = 1 \). Therefore, it is possible to derive the maximum fractional error in \( w_i \) and \( w_j \) due to quantisation. The maximum fractional error in a variable \( v \) due to quantisation is defined as

\[
\varepsilon_{vq} = \frac{v_{ss} - v_{sq}}{v_{ss}}
\]

Using the above definition, the maximum fractional error in \( w_i \) and \( w_j \) is derived as:

\[
\hat{\varepsilon}_{w_{1q}} = \frac{2^b}{d_{ll}} \quad \text{and} \quad \hat{\varepsilon}_{w_{2q}} = \frac{2^b}{d_{ll}} \quad (3.27)
\]
Alternatively, if \( d_{i1} < d_{i3} \), then
\[
\hat{e}_{w_{1q}} = \frac{2^b}{d_{i2}} \quad \text{and} \quad \hat{e}_{w_{2q}} = \frac{2^b}{d_{i2}}
\] (3.28)

Re-working assuming the results of the multiplication's are rounded rather than truncated gives
\[
\hat{e}_{w_{1q}} = \frac{2^{(b+1)}}{d_{i1}} \quad \text{and} \quad \hat{e}_{w_{2q}} = \frac{2^{(b+1)}}{d_{i1}} \quad \text{for} \quad d_{i1} > d_{i2}
\] (3.30)

The above expressions have identified errors in the internal variables, and these can be used to predict errors in the output \( y \) due to quantisation, including its fractional error \( e_{yq} \) and maximum fractional error. For the case of \( d_{i1} > d_{i2} \), and assuming truncation, the fractional error can be derived as
\[
0 \leq e_{yq} < \left[ \frac{(p_1 + p_2) - (q_1 + q_2)}{q_1 + q_2} \right] \frac{2^b}{d_{i1}}
\]
and the maximum fractional error is, therefore, given by
\[
e_{yq} \leq \left[ \frac{(q_1 + q_2) - (p_1 + p_2)}{q_1 + q_2} \right] \frac{2^b}{d_{i1}}
\]

Similar expressions can be derived for \( d_{i1} > d_{i2} \).

The above equations were derived assuming the controller had complex poles. Similar expressions can also be derived if the controller had a pair of real repeated roots.

Rearranging equation (3.29) (or (3.30) if \( d_{i2} > d_{i1} \)) to express the number of fractional bits as a function of the maximum fractional error in the internal variables due to rounding gives
\[
b = \log_2 \left( \frac{1}{d_{i1} \hat{e}_{w_{1q}}} \right) - 1
\]
This can be used to evaluate the number of fractional (or underflow) bits needed to ensure that the error in the internal variables does not exceed the maximum value that can be tolerated.
As for the modified canonic structure, the efficiency of the two methods in predicting the underflow wordlength has to be determined. Once again a 10 Hz notch filter with a damping factor of 0.05 and a lowpass filter with a damping factor of 0.707 are implemented. Both filters are examined for sample frequencies of 100 Hz, 500 Hz, 1 kHz and 10 kHz. Also, suppose the maximum fractional error that can be tolerated is 5%. The number of underflow bits predicted using the two methods are given in Table 3.6.

<table>
<thead>
<tr>
<th>Sample Frequency (Hz)</th>
<th>Low pass Filter</th>
<th>Notch Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Round-off</td>
<td>Steady State</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>500</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>10000</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3.6  Underflow bits as predicted by round-off noise and steady state analysis for the cross coupled structure

It can be seen from the table that there is a discrepancy between the predictions from the two methods. Additionally, the difference increases as the sample frequency is increased with respect to the filter frequency and the difference is quite large at high sample frequencies. Since the impact of the difference on the output is not immediately obvious, simulation is once again used to compare and contrast the results.

The response of the filters to a step input is examined using the underflow bit predictions given in Table 3.6. Figures 3.23 and 3.25 show the results for the low pass filter for sample frequencies of 500 Hz and 10 kHz respectively and Figures 3.26 and 3.28 show the results for the notch filter, again for sample frequencies of 500 Hz and 10 kHz.
From Figure 3.23 it is not clear whether the error in the output, due to the underflow bits predicted by the steady state and round-off noise analysis, is less than the maximum allowable limit of 5%. A graph of the error in the output is therefore plotted in Figure 3.24. From this graph it is clear that the error from both is less than 5%.

However, when the sample frequency is increased to 10 kHz, Figure 3.25 clearly shows that the number of underflow bits predicted by the round-off noise analysis is not enough as the error in the output, about 40%, far exceeds the maximum allowable value of 5%. The steady state analysis results, on the other hand, give a much better prediction.

![Graph showing step response of a 10 Hz low pass filter sampled at 500 Hz](image)

*Figure 3.23 - Step response of a 10 Hz low pass filter sampled at 500 Hz*
Figure 3.24 - Error in the output of a 10 Hz low pass filter sampled at 500 Hz

Figure 3.25 - Step response of a 10 Hz low pass filter sampled at 10 kHz
From Figure 3.26 it can be seen that for the notch filter sampled at 500 Hz the number of bits as predicted by the round-off noise analysis are not enough while the number predicted by the steady state analysis are enough. A check of the error plot, shown in Figure 3.27, confirms this.

However, when the sample frequency is increased, see Figure 3.28, the number of bits as predicted by the round-off noise analysis are not enough while the number predicted by the steady state analysis are enough. A plot of the error in the output for this case is shown in Figure 3.29.

In Figures 3.26 and 3.28, a limit cycle oscillation situation arises. This is generally not a problem as long as the magnitude of the oscillation does not perturb the system under control. In the case of Figures 3.26 and 3.28, the size of the oscillation is within the acceptable limit.

![Figure 3.26 - Step response of a 10 Hz notch filter sampled at 500 Hz](image)
Structures for Implementation

Roundoff Noise (5 bits)

Steady State (7 bits)

Figure 3.27 - Error in Output of a 10 Hz notch filter sampled at 500 Hz

Figure 3.28 - Step response of a 10 Hz notch filter sampled at 10 kHz
The conclusions regarding the analyses methods are similar to those obtained for the cascaded modified canonic structure: That is, in general the round-off noise analysis is overly optimistic and does not efficiently predict a sufficient wordlength for the internal variables, especially at high sample rates; the steady state analysis is, on the other hand, a more reliable method and efficiently predicts a sufficient wordlength for the internal variables, sometimes rather conservatively.

The difference between the predictions of the two methods is of course due to the principles which the methods use in predicting the wordlengths:

- the steady state analysis explicitly gives worst case and is therefore pessimistic in its predictions,
- while in the round-off noise analysis a judgement, such as the one made in equation (3.13a) \(2\sigma = \text{r.m.s error}\), is made to ensure that the number of bits satisfy the r.m.s error condition. If the judgement made is not adequate then the result will be an overly optimistic prediction.
3.4 Comparison of the Structures

As has been mentioned in section 3.3, the cross coupled structure was devised to take advantage of the lower wordlength requirements of first order sections while still being able to implement second order functions. Since the wordlength analysis of both the cascaded modified canonic and cross coupled structures have been performed, the two structures can be compared in terms of their wordlength requirements. To do this consider the implementation of a 10 Hz low pass filter with $\zeta = 0.707$, and a 10 Hz notch filter with $\zeta = 0.05$ and a sample frequency of 1 kHz for both. Also, assume an input/output wordlength of 12 bits is used.

Low pass filter

Using the steady state analysis results, Table 3.4 shows that 9 underflow bits need to be provided for - giving a total wordlength of 21 bits for the internal variables for the modified canonic structure. For the cross coupled structure, Table 3.7 shows that 8 underflow bits tells need to be provided for - giving a total word length of 20 bits for the internal variables. So using the cross coupled structure a saving of 1 bit is realised.

Notch filter

For the notch filter the total wordlength of the internal variables for the modified canonic structure works out to be 22 bits while the total for the cross coupled structure works out to be 20 bits. Once again a saving is realised using the cross coupled structure. This time it is 2 bits.

The above comparison's have been made using the results of the steady state analysis which are known not to be very precise: it tends to be pessimistic and over-predict. On the other hand, the round-off noise analysis cannot be used as it tends to be overly optimistic and hence under-predicts the number of underflow bits necessary. Therefore, should the above conclusions be regarded as final? It was decided to work out the r.m.s errors for each of the structures from simulations for each of the structures. In order to use the r.m.s errors, the input to the filter requires...
It is imperative that the input be such that the operating conditions of the filter are captured. For example if, a step input was to be used then the r.m.s error would be dependent on how long the simulation was executed for. If the simulation time was too great then the r.m.s error would be dominated by the steady state value of the output and the transient behaviour of the filter would be over shadowed; if the time was too short then there is a danger that not enough information would have been gathered to make any reasonable conclusions. Therefore, a pseudo-random binary sequence (PRBS) was used as an input. The PRBS has the property that it can be repeated and is also random in nature and contains energy at a wide range of frequencies which exercises the filter more effectively.

The sequence was structured such that it would repeat itself every second (implying a minimum frequency of 1Hz) and the sequence would have 50 binary numbers (implying a maximum frequency of 50 Hz). The filter was simulated with the number of underflow bits varying between 4 and 13 (one less and greater than the minimum and maximum values, as predicted in Tables 3.4 and 3.6, respectively). The results for a 10Hz notch ($\zeta = 0.05$) and lowpass ($\zeta = 0.707$) filter are given in Figures 3.30 and 3.31 respectively.
Figure 3.30 - Variation of r.m.s error with number of underflow bits for a 10 Hz Notch Filter sampled at 500 Hz and 10 kHz.

Figure 3.31 - Variation of r.m.s error with number of underflow bits for a 10 Hz Lowpass Filter sampled at 500 Hz and 10 kHz.
If an r.m.s error value of 0.05 is used as the cut-off value, then Figure 3.30 indicates that the cross coupled structure requires less underflow bits than the equivalent modified canonic structure for the same sample frequency. For example, when the sample frequency is 500 Hz, with 5 underflow bits, the cross coupled structure produces an error of around 0.037, which is within the cut-off limit above, while the cascaded modified canonic structure has an error of 0.11, which is unacceptable. From the graph it can be seen that the modified canonic structure requires 6 underflow bits to get the error within the acceptable bound. When the sample frequency is increased, the cross coupled structure requires 10 underflow bits while the modified canonic structure requires 11 underflow bits.

For the lowpass filter, Figure 3.31 indicates that the two structures are evenly matched. At a sample frequency of 500 Hz, both structures require 8 underflow bits to be within the 0.05 acceptable bound while at a sample frequency of 10 kHz, both structures require 12 underflow bits to be within the acceptable limit.

Unfortunately, the saving using the cross coupled structure is not as large as would be expected from a structure which is inherently first order in nature. It was a reasonable hypothesis that the first order cross-coupled structure would give an improvement but how much was not clear. In fact when the saving is offset against the increased number of multiplication's (8 for cross coupled as opposed to 5 for modified canonic) it hardly seems to be worth it. However all is not lost, because if the controller was to be implemented on a parallel processor, the execution time for the cross coupled structure could be shorter than the time it would take to execute the cascaded modified canonic structure.
Two important aspects of digital controller implementation have been presented in this chapter:

1. Internal variable wordlength requirements and
2. Structures for implementation

Consider first the wordlength issue. To accurately determine the wordlength of the variables in terms of underflow, two different methods were studied:

1. the steady state analysis and
2. the round-off noise analysis

For the cascaded modified canonic structure, the steady state analysis is not presented in detail as it covered in the literature but the round-off noise analysis is presented in some detail, in section 3.2.2.1, as it has not been previously used this structure. A comprehensive analysis of the two methods to determine which of the two methods is suitable for predicting the number of underflow bits accurately has been carried out. Method 1 above was shown to be pessimistic due to the basis of assuming a worst case condition, thereby tending to over-predict the number of underflow bits, while Method 2 is overly optimistic, tending to under-predict the number of underflow bits, and usually relies on having to make a judgement ($2\sigma = \text{error}$).

From the structural point of view, a novel first order structure - the cross coupled modified canonic structure - has been presented. The structure was devised with the objective of taking advantage of its first order nature to obtain internal variable wordlengths which would be less than the equivalent second order cascaded modified canonic structure.

A detailed wordlength analysis, both overflow and underflow, has been carried out in sections 3.3.2 and 3.3.3 respectively. As with the cascaded modified canonic structure, it was shown that the overflow of the internal variables is not an issue. Additionally, a detailed underflow bits analysis has been carried out.
and once again both a steady state analysis and a round-off noise analysis of the structure has been carried out to determine which would accurately predict the underflow bits requirement. A comparison of the two analyses methods confirms the results obtained with the cascaded modified canonic structure.

Initial results of the comparisons of the two structures have indicated that the resulting wordlengths do not give as great a saving as would be expected. To compare the structures, it was decided that simulation would be used as the analytical methods were unreliable. For the simulation purposes a PRBS was used as an input to analyse the variation in r.m.s error, of the filter output, against a changing number of underflow bits.
The modified canonic structure presented in Chapter 3 is robust to changes in coefficients and has superior wordlength requirements compared to the canonic z-operator structure [Forsythe and Goodall (1991)]. Due to the superior performance of the structure, it would be sensible to use it when implementing controllers. Since the structure is represented as a transfer function, any controller which is specified as a transfer function can easily be implemented as a cascade or parallel combination of first and second order sections based on the modified canonic form. However, what if the controller is specified in state space form?

One possible solution is to convert the controller to transfer function representation and then implement it as a combination of first or second order modified canonic sections. This, however, is not the ideal solution and hence it would be better if an algebraic algorithm could be used to convert between the structures.

Presented in this chapter is an algorithm which can be used to convert a standard discrete state space system based on the z-operator to a modified canonic state space system. At present the algorithm can only handle single input systems and extensions to multi input systems need to be made.

To test the coefficient sensitivity of the structures a frequency response magnitude sensitivity analysis of the state space system is performed. Using this analysis the z-operator state space formulation is compared with the δ-operator formulation based on the modified canonic form.
4.1 State Space Algorithms

Various different state space algorithmic structures exist depending on the controller design method. Examples of the different structures can be found in [Hanselmann (1987)]. Some of these different structures cannot be reduced to the standard state space form given in equation (4.1) because of the presence of "current" estimator term. The standard state space form can be used to represent a dynamic system with a number of inputs and outputs.

\[
\begin{align*}
X' &= AX + BU \\
y &= CX + DU
\end{align*}
\]

(4.1)

Such a system may also appear as a subsystem in a complex controller. Its input thus does not necessarily coincide with the plant measurement, reference and measured disturbance vectors, and its output is not necessarily the control input vector to the plant. The usual convention of \( u \) being the input and \( y \) being the output of this system has therefore been adopted. Note the use of subscript \( z \) to denote that this system has been derived using \( z \)-operator formulation.

4.2 The state space modified canonic structure

The second order modified canonic structure shown in figure 3.2 is based on the \( \delta \)-operator. This structure is a single input single output (SISO) system and the internal variables \( w \) and \( x \) can be taken to be states of the system. The equation to calculate the output \( y \) at time \( k \) is given by

\[
y_k = p_1 v_k + p_2 w_k + p_3 x_k
\]

(4.2)

but \( v_k \) is calculated as

\[
v_k = u_k - w_k - x_k
\]

(4.2a)

substituting (4.2a) into (4.2) and rearranging, the following can be shown

\[
y_k = \left[ (p_2 - p_1) (p_3 - p_1) \right] \begin{bmatrix} w_k \\ x_k \end{bmatrix} + p_1 u_k
\]

(4.3)

The equations to update the values of the internal variables (or states of the
The Modified Canonic State Space Structure

system) are

\[ w_{k+1} = w_k + d_1 v_k \]
\[ = w_k + d_1 (u_k - w_k - x_k) \]
\[ = w_k (1 - d_1) - d_1 x_k + d_1 u_k \]  \hspace{1cm} (4.4)

and

\[ x_{k+1} = x_k + d_2 w_k \]  \hspace{1cm} (4.5)

Equations (4.4) and (4.5) can be written as

\[ w_{k+1} = (1 - d_1) w_k - d_1 x_k + d_1 u_k \]
\[ x_{k+1} = d_2 w_k + x_k \]  \hspace{1cm} (4.6)

Comparing (4.6) and (4.3) with (4.1), it can be seen that the modified canonic structure can also be represented as a standard state space system with

\[ A_\delta = \begin{bmatrix} 1 - d_1 & -d_1 & -d_1 \\ d_2 & 1 & 0 \\ 0 & d_3 & 1 \end{bmatrix}, \quad B_\delta = \begin{bmatrix} d_1 \\ 0 \\ 0 \end{bmatrix}, \quad C_\delta = \begin{bmatrix} (p_2 - p_i) (p_3 - p_i) (p_4 - p_i) \end{bmatrix}, \quad D_\delta = p_i \]

Note the subscript \( \delta \) will be used from now to distinguish between the representation for the modified canonic state space structure and the representation of equation (4.1). Similarly, any higher order modified canonic structure can also be represented in standard state space form. For example for a 3rd order modified canonic structure, the \( A_\delta, B_\delta, C_\delta \) and \( D_\delta \) are

\[ A_\delta = \begin{bmatrix} 1 - d_1 & -d_1 & -d_1 \\ d_2 & 1 & 0 \\ 0 & d_3 & 1 \end{bmatrix}, \quad B_\delta = \begin{bmatrix} d_1 \\ 0 \\ 0 \end{bmatrix}, \quad C_\delta = \begin{bmatrix} (p_2 - p_i) (p_3 - p_i) (p_4 - p_i) \end{bmatrix}, \quad D_\delta = p_i \]

and for a 4th order

\[ A_\delta = \begin{bmatrix} 1 - d_1 & -d_1 & -d_1 & -d_1 \\ d_2 & 1 & 0 & 0 \\ 0 & d_3 & 1 & 0 \\ 0 & 0 & d_4 & 1 \end{bmatrix}, \quad B_\delta = \begin{bmatrix} d_1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad C_\delta = \begin{bmatrix} (p_2 - p_i) (p_3 - p_i) (p_4 - p_i) (p_5 - p_i) \end{bmatrix}, \quad D_\delta = p_i \]  \hspace{1cm} (4.6a)

The trend for higher order structures is obvious. So in general the modified canonic structure can also be represented as a state space system of the form

Architectural Considerations for a CSP
\[
\begin{align*}
    x_{k+1} &= A_\delta x_k + B_\delta u_k \\
    y_k &= C_\delta x_k + D_\delta u_k
\end{align*}
\] (4.7)

where as before the \( x \) vector contains the states of the system and \( u \) represents the input while \( y \) represents the output of the system. The problem one is faced with is of transforming a set of matrices \([A_z, B_z, C_z, D_z]\) to another set of matrices \([A_\delta, B_\delta, C_\delta, D_\delta]\). This transformation can be achieved by means of a similarity transform. If one assumes that the transformation is achieved by a matrix \( T \) then the matrices \([A_\delta, B_\delta, C_\delta, D_\delta]\) are related to the matrices \([A_z, B_z, C_z, D_z]\) as shown below

\[
\begin{align*}
    A_\delta &= T^{-1} A_z T, \\
    B_\delta &= T^{-1} B_z, \\
    C_\delta &= C_z T, \\
    D_\delta &= D_z
\end{align*}
\]

From the representation of the modified canonic system above note that \( A_\delta \) and \( B_\delta \) must be of a certain form. Additionally note that after transformation the two systems are similar and therefore the eigenvalues of both system must be the same. This property can be used to calculate the coefficients of \( A_\delta \) and \( B_\delta \) and hence form the two matrices.

4.3 Eigenvalues

The eigenvalues of equations (4.1) and (4.7) can be found by solving the characteristic equations \( |\lambda I - A_z| = 0 \) and \( |\lambda I - A_\delta| = 0 \) respectively. For simplicity let us assume that the system of (4.1) is in controller canonical form [Ogata (1987)]. This means that for an nth order system the \( A_z \) matrix is of the form:

\[
A_z = \begin{bmatrix}
-a_1 & -a_2 & -a_3 & \cdots & -a_{n-1} & -a_n \\
1 & 0 & 0 & \cdots & 0 & 0 \\
0 & 1 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 1 & 0
\end{bmatrix}
\] (4.8)

It can be shown that the characteristic equation for an nth order system then works out to be
For equation (4.7) it can be shown that for an nth order system,

\[ |\lambda I - A_s| = (\lambda - 1)^n + d_1(\lambda - 1)^{n-1} + d_1d_2(\lambda - 1)^{n-2} + \ldots + d_1d_2\ldots d_n(\lambda - 1) + d_1d_2\ldots d_n \]  

(4.10)

By equating equations (4.9) and (4.10), an expression relating \([a_1, a_2, \ldots, a_n]\) and \([d_1, d_2, \ldots, d_n]\) can be formulated. The process is not straightforward and the solution does not just 'drop out'. The relationship is:

\[ d_n = [T_d'] \left( a_n - c_n \right) \]  

(4.11)

where

\[
\begin{bmatrix}
  a_1 \\
  a_2 \\
  a_3 \\
  \vdots \\
  a_n
\end{bmatrix}
\]

This relationship was arrived at by examining the trends through different orders of systems: that is for 2nd, 3rd, 4th, 5th, etc. The resulting algebraic expression for the different orders had a common format which could be grouped into the form given in equation (4.11) above. Further details are not included in the main text and can be found in Appendix E.

In equation (4.11), \([T_d']\) is a \(n \times n\) matrix and \(c_n\) is a \(n \times 1\) column vector. The forms of \([T_d']\) and \(c_n\) for a nth order system are given in Appendix E. From this it is seen that for any order of system, \([T_d']\) and \(c_n\) can be derived. So using the results of equation (4.11), the coefficients of \(A_{\delta}\) and \(B_{\delta}\) can be calculated. But what about \(C_{\delta}\)?

To calculate the coefficients of \(C_{\delta}\), the transformation matrix \(T\) which will transform the controller canonical form to the modified canonic structure needs to be calculated. In order to calculate \(T\) an understanding of how a system can be transformed to the controller canonical form is required.
4.4 Transformations to controller canonical form

Any system defined by

\[
\begin{align*}
    x_{k+1} &= G x_k + H u_k \\
    y_k &= C x_k + D u_k
\end{align*}
\]  

(4.12)
can be transformed to the controller canonical form by means of a transformation matrix [Ogata (1987)]

\[
    T = MW
\]  

(4.13)

where

\[
    M = [H : GH : G^2 H : \cdots : G^{n-1} H]
\]  

(4.14)

and

\[
    W = \begin{bmatrix}
        a_{n-1} & a_{n-2} & \cdots & a_1 & 1 \\
        a_{n-2} & a_{n-3} & \cdots & 1 & 0 \\
        \vdots & \vdots & \ddots & \vdots & \vdots \\
        a_1 & 1 & \cdots & 0 & 0 \\
        1 & 0 & \cdots & 0 & 0
    \end{bmatrix}
\]  

(4.15)

The elements \(a_i\) shown in \(W\) are the coefficients of the characteristic equation

\[
    |\lambda I - G| = \lambda^n + a_1 \lambda^{n-1} + a_2 \lambda^{n-2} + \cdots + a_{n-1} \lambda + a_n = 0
\]

Applying the transformation \(T\) to the system in (4.12), the states \(x(k)\) are being transformed to, say, \(z(k)\) and therefore the resulting system can be shown to be

\[
\begin{align*}
    x_{k+1} &= \hat{A} x_k + \hat{B} u_k \\
    y_k &= \hat{C} x_k + D u_k
\end{align*}
\]  

(4.16)

where

\[
    \hat{A} = T^{-1} G T = (MW)^{-1} G(MW) = W^{-1} M^{-1} GMW
\]  

(4.17)

\[
    \hat{A} = \begin{bmatrix}
        0 & 1 & 0 & \cdots & 0 \\
        0 & 0 & 1 & \cdots & 0 \\
        \vdots & \vdots & \vdots & \ddots & \vdots \\
        0 & 0 & 0 & \cdots & 1 \\
        -a_n & -a_{n-1} & -a_{n-2} & \cdots & -a_1
    \end{bmatrix}
\]

and

Architectural Considerations for a CSP 98
The modified canonic state space structure

\[
\hat{B} = T^{-1}H = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, \quad \hat{C} = CT
\]

The above analysis can be used if the system of equation (4.12) is assumed to be the modified canonic structure and hence it can be converted to the controller canonical form by means of the transformation \( T \). So to reverse the process, that is to convert the controller canonical form to our modified canonic system, one would have to apply the transformation \( T' \). Finally, note that for the analysis in section 4.3, the starting point was the controller canonical form as given in equation (4.8) but the analysis in this section assumes the controller canonical form as given in equation (4.17). Equation (4.17) can be converted to (4.8) by means of an additional transformation matrix

\[
R = \begin{bmatrix}
0 & 0 & \ldots & 0 & 1 \\
0 & 0 & \ldots & 1 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
1 & 0 & \ldots & 0 & 0
\end{bmatrix}
\]

that is

\[
A_z = R^{-1}\hat{A}R \quad \text{and} \quad B_z = R^{-1}\hat{B}
\]

substituting for \( \hat{A} \) and \( \hat{B} \) gives

\[
A_z = R^{-1}T^{-1}GTR \quad \text{and} \quad B_z = R^{-1}T^{-1}H \tag{4.18}
\]

From equation (4.18) the transformation needed to convert from the modified canonic system to the controller canonical form of equation (4.8) is

\[
\hat{T} = TR \tag{4.19}
\]

So the transformation which can be used to convert the controller canonical form of (4.8) to the modified canonic form is

\[
\hat{T}^{-1} = (TR)^{-1} = R^{-1}T^{-1} \tag{4.20}
\]

Once the transformation matrix is known, \( C_δ \) can be calculated.
The Modified Canonic State Space Structure

4.5 The algorithm

4.5.1 The procedure

The results of the analysis performed in the previous sections can be summarised as follows: the analysis performed above works for a Single Input Single Output (SISO) system and the steps involved in transforming a state space system expressed in controller canonical form to one expressed in modified canonic form are:

1. Form the $G (= A_δ)$ and $H (= B_δ)$ matrices of (4.12) using equation (4.11). The $[T_ε^{-1}]$ and $C_ε$ matrices can be formed by referring to Appendix E. This will give the coefficients $d_v, d_y, ..., d_n$.
2. Solve for $M$ using equation (4.14).
3. Calculate $W$ using equation (4.15).
5. Find $\hat{T}$ using (4.19) and therefore $\hat{T}^{-1}$. Using $\hat{T}^{-1}, C_δ$ can be calculated and $D_δ = D_ε$.

For a Single Input Multiple Output (SIMO) the $A_δ$ and $B_δ$ matrices are derived as before and the difference is in the derivation of the Output Matrix $C$. The $C$ matrix 'picks' out the appropriate states and combines them to form the relevant outputs. Fortunately, for the SIMO case, the procedure does not need to be altered and the last step will form the appropriate $C_δ$ matrix.

The algorithm has been coded as a MATLAB function which is given in Appendix F. The resulting modified canonic structure was simulated and the results of the simulation were compared with the results of the equivalent controller canonical structure.
4.5.2 Testing

To validate the algorithm, filters of different orders were simulated and the step responses of the two realisations compared. These filters were first formulated as continuous transfer functions and the equivalent discrete forms were derived using the bilinear transformation (using the \textit{c2dm} function in MATLAB). A state space formulation of these transfer functions can now be obtained (using the \textit{tf2ss} function in MATLAB) with the filter (or controller) being expressed in controller canonical form. The algorithm described in section 4.5.1 can now be used to obtain an equivalent representation of the filters in modified canonic form. For all the filters examined, the algorithm works and the resulting step responses for the \textit{z}-operator formulation and the \textit{\delta}-operator modified canonic form are identical.

4.5.3 Beware

So far it has been assumed that the discrete controller will be in controller canonical form. What happens when the controller is not? One can easily obtain a controller canonical representation of an equivalent discrete system as has been seen in section 4.4. However, this method is numerically ill conditioned when the eigenvalues are close to each other. Due to the ill conditioning the resulting controller canonical system is sometimes 'wrong' and will not perform its intended function appropriately. This can have disastrous consequences for a control system.

So even though an algorithm has been found to convert between controller canonical formulations and the modified canonic formulation, one is in a position whereby it cannot always be used. So is there another method which when used will produce the modified canonic system and be numerically stable as well? Fortunately there is a method and in order to appreciate it, the issue of implementing state space controllers which are specified either in continuous or discrete time needs to be considered.
4.6 Implementing state space controllers

4.6.1 Continuous time models

If the control design is done in continuous time then the state space model is of the form

\[
\begin{align*}
    \dot{x} &= Ax + Bu \\
    y &= Cx + Du
\end{align*}
\]  \hspace{1cm} (4.21)

An equivalent discrete system of the form given in equation 4.1 can be evaluated. The evaluation involves calculating the matrix exponential and a number of methods can be used to perform this operation [Middleton and Goodwin (1990)]. Some of these methods are numerically ill conditioned. The resulting system is a z-operator model and can be written as

\[
\begin{align*}
    zX &= A_z X + B_z u \\
    Y &= C_z X + D_z u
\end{align*}
\]  \hspace{1cm} (4.22)

The equivalent \( \delta \)-operator model could be obtained by substituting \( z=\delta+1 \) in equation 4.22. Even though this model will be technically correct, this is not the best way to evaluate the \( \delta \)-operator model, since most of the numerical problems associated with the evaluation of the z-operator model are carried over to the \( \delta \)-operator model.

A better method is to evaluate the \( \delta \)-operator model directly from the continuous time state space equations [Middleton and Goodwin (1990), pp47]. The resulting system can be written as

\[
\begin{align*}
    \delta X &= A_\delta X + B_\delta u \\
    Y &= C_\delta X + D_\delta u
\end{align*}
\]  \hspace{1cm} (4.23)

Note that the definition of the \( \delta \)-operator - \( \delta=(z-1)/T_s \) - used by Middleton and Goodwin is different from the definition - \( \delta=z-1 \) - used by the author. So to obtain a system which conforms to the author's definition of \( \delta \), the \( A_\delta \) and \( B_\delta \) matrices need to be multiplied by \( T_s \). The above procedure produces numerically stable systems. Therefore, one can now attempt to express the system in modified
4.6.2 Transforming to modified canonic form

If the system in equation (4.23) is not in canonical form, it can easily be expressed in that form without fear of obtaining an 'incorrect' equivalent due to ill conditioned matrices. In MATLAB this can be done by running the ss2tf function (gives transfer function equivalent of the state space system) followed by tf2ss (the tf2ss function produces a state space equivalent which is in canonical form). Note that by canonical form, the matrices of equation (4.23) are assumed to be in the following form:

\[
A_B = \begin{bmatrix}
-a_1 & -a_2 & -a_3 & \cdots & -a_{n-1} & -a_n \\
1 & 0 & 0 & \cdots & 0 & 0 \\
0 & 1 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 1 & 0
\end{bmatrix}
\]  

(4.24)

where the coefficients \(a_1, a_n\) are the feedback path coefficients. So the next stage is to get from the canonic form to the modified canonic form. The modified canonic structure is such that the feedback coefficients are made equal to 1 so that the states have very similar maximum values. This can be thought of as transforming from one set of states to another set. If say \(X_c\) represents the canonic form states and that \(X_{mc}\) represents the modified canonic states, then they are related by

\[X_c = TX_{mc}\]

where \(T\) is the transformation matrix such that all the modified canonic states have similar values. It can be shown that by using
The states for the canonic form are transformed to the modified canonic form and the resulting modified canonic form model is given as:

\[ \delta \mathbf{x}_{mc} = \mathbf{A}_{\delta mc} \mathbf{x}_{mc} + \mathbf{B}_{\delta mc} u \]
\[ \mathbf{y} = \mathbf{C}_{\delta mc} \mathbf{x}_{mc} + \mathbf{D}_{\delta mc} u \]

where

\[ \mathbf{A}_{\delta mc} = \mathbf{T}^{-1} \mathbf{A}_{\delta} \mathbf{T} ; \mathbf{B}_{\delta mc} = \mathbf{T}^{-1} \mathbf{B}_{\delta} ; \mathbf{C}_{\delta mc} = \mathbf{C}_{\delta} \mathbf{T} ; \mathbf{D}_{\delta mc} = \mathbf{D}_{\delta} \]

### 4.6.3 Example

To understand the alternative method outlined in section 4.6.2, consider the implementation of a 7th order \( H_\infty \) controller designed in [Tsai (1989)]. Though the controller is designed for a MIMO system, for simplicity, only a single input single output case is considered and the \( A, B, C \) and \( D \) matrices are as given in Appendix G. This controller is designed in the continuous domain and therefore needs to be converted to a discrete equivalent. Additionally, the controller is a good example of the numerical problems mentioned in section 4.5.3; that is if the algorithm described in section 4.5.1 were to be used to produce the modified canonic form, then problems with ill conditioned matrices will arise as the eigenvalues of the controller are close to each other.

The controller is therefore converted to the discrete form by using the method of [Middleton and Goodwin (1990), pp47] with the sample frequency equal to 1kHz; that is \( T_s \) is set to 1ms. (Note that the \( e2del \) function in the delta toolbox in...
MATLAB performs this operation.) This gives a system which is of the form given in equation (4.23) with the $A_δ$ matrix not in canonical form. The $A_δ$ and $B_δ$ matrices are multiplied by $T$, to obtain a model for which $δ=z-1$. At this stage, the $A_δ$ matrix is not in the controller canonical form of equation (4.24) and to obtain this, the ss2tf function of MATLAB is used followed by the tf2ss function.

With the $A_δ$ matrix expressed in controller canonical form, the transformation matrix $T$ is then formulated and equation (4.26) is used to obtain the modified canonical form. The results at each of the stages are also given in Appendix G. Simulations show that the resulting controller in modified canonical form is identical to the original controller. A flowchart depicting the above is shown in Figure 4.1.

![Flowchart](image)

**Figure 4.1 - Flowchart depicting process to convert from a continuous state space to a modified canonic state space system**
4.6.4 Discrete models

There are cases when the controller is designed in the digital domain and the resulting controller will therefore be expressed as shown in equation (4.22). Assuming that the controller has no numerical problems associated with it, the controller should first be expressed in controller canonical form and the algorithm described in section 4.5.1 could then be used to derive the modified canonic representation easily. If the controller is numerically unstable then the best that can be done is to produce the equivalent δ form by substituting \( z = 1 + 1 \) to get a system which is more robust.

4.7 Sensitivity Analysis

Coefficient sensitivity has been known to be a problem at high sample rates. As was seen in Chapter 2, sensitivity analysis is usually performed to assist in the determination of the wordlengths of the coefficients. Additionally, the analysis also provides information relating to the sensitivity of the system to changes in coefficients. Therefore, in this section the issue of coefficient sensitivity is examined: that is, how sensitive is the state space system to changes in the coefficients of the \( A, B, C \) and \( D \) matrices?

To perform a sensitivity analysis, a possible method that can be used is to find a coefficient sensitivity matrix which will relate fractional changes in the discrete coefficients to fractional changes in the continuous coefficients [Forsythe and Goodall (1991)]. This method is straightforward for simple first and second order systems represented as transfer functions, but is algebraically complex when extensions to a state space system are made. To overcome this, the matrix could be derived using simulation.

An alternative method which can be used is to test the sensitivity of the magnitude of the frequency response of the state space system to changes in the coefficients. This is the method that is presented in this section. The method is semi-analytical and can be coded quite easily in software. The results of the
The Modified Canonic State Space Structure

Sensitivity analysis can be used to interpret how inaccuracies caused by quantisation in the coefficients of the discrete state space controller will affect the overall performance of the closed loop control system. These results can be used to determine the precision to which the coefficients need to be represented to achieve performance to a certain accuracy.

4.7.1 Magnitude sensitivity

The transfer function of the state space system described by equation (4.22) can be shown to be

$$H(z) = C_z(zI - A_z)^r B_z + D_z$$

and the magnitude of the frequency response can be calculated by substituting $z = e^{io\theta}$. A small change in the transfer function is given by the following standard mathematical expression

$$\frac{\delta H(z)}{H(z)} = \frac{\partial H(z)}{\partial A_z} \delta A_z + \frac{\partial H(z)}{\partial B_z} \delta B_z + \frac{\partial H(z)}{\partial C_z} \delta C_z + \frac{\partial H(z)}{\partial D_z} \delta D_z$$

which can be re-written in terms of fractional changes as

$$\frac{\delta H(z)}{H(z)} = \left(\frac{\partial H(z)}{\partial A_z} \frac{A_z}{H(z)}\right) \delta A_z + \left(\frac{\partial H(z)}{\partial B_z} \frac{B_z}{H(z)}\right) \delta B_z + \left(\frac{\partial H(z)}{\partial C_z} \frac{C_z}{H(z)}\right) \delta C_z + \left(\frac{\partial H(z)}{\partial D_z} \frac{D_z}{H(z)}\right) \delta D_z$$

(4.28)

Note that in the above expression for fractional changes all the operations (multiplications and divisions) are on an element by element basis and are not matrix operations. By substituting $z = e^{io\theta}$ in equation (4.28) and taking the magnitude, gives the magnitude sensitivity:

$$\left| \frac{\partial H(e^{io\theta})}{\partial A_z} \frac{A_z}{H(e^{io\theta})} \right|$$

That is, at each frequency point a matrix of sensitivity factors which relates fractional changes in coefficients $a_{ij}$ (i=1..n, j=1..n, n=order of system) to fractional changes in the transfer function magnitude is obtained. Hence a number of sensitivity factor matrices, depending on the frequency points under
consideration, are created. The r.m.s. value of these sensitivity factors over the representative frequency range has been calculated to obtain a single figure of merit for each coefficient $a_{ij}$ (N.B. other figures of merit could easily be derived, for example the worst case variation).

To calculate the partial derivative matrices of equation (4.28), the results given in [Li and Gevers (1993)] can be used:

$$\frac{\partial H(z)}{\partial A_z} = (zI - A_z^T)^l C_z B_z^T (zI - A_z^T)^l$$  (4.29)

This is a $n \times n$ matrix where $n$ is the number of states (or the order of the system)

$$\frac{\partial H(z)}{\partial B_z} = (zI - A_z^T)^l C_z$$  (4.30)

This is a $n \times m$ matrix where $m$ is the number of inputs

$$\frac{\partial H(z)}{\partial C_z} = (zI - A_z^T)^l B_z$$  (4.31)

This is a $k \times n$ matrix where $k$ is the number of outputs

$$\frac{\partial H(z)}{\partial D_z} = 1$$  (4.32)

This is a $k \times m$ matrix.

### 4.7.2 δ-operator models

The analysis above assumed that the model is in $z$-operator form. For the $δ$-operator model the original system is as given in equation (4.26) and the transfer function is therefore given by

$$H(δ) = C_δ (δ I - A_δ^T)^l B_δ + D_δ$$

and the remaining equations (4.28 - 4.32) are identical with the $z$'s replaced by $δ$'s and the magnitude can be obtained by substituting $δ= e^{j\omega t} - 1$. 

Architectural Considerations for a CSP
4.7.3 Choosing a frequency range

Since the magnitude of the frequency response is being considered, it is vital to choose a frequency range such that the sensitivity factors are not biased by either the low frequencies or the high frequencies. A sensible choice is to calculate the magnitude, and hence the sensitivity factors, from a decade below the lowest relevant frequency to a decade above the maximum relevant frequency. The relevant frequencies can be derived directly from the eigenvalues of the system. For a discrete system if \( z = a \) is an eigenvalue of the system then the relevant frequencies are calculated as:

\[
\begin{align*}
z &= e^{j\omega r} = a \Rightarrow \omega = \frac{1}{T} \log_e a \\
\Rightarrow \omega &= \left| \frac{1}{T} \log_e a \right|
\end{align*}
\]

4.7.4 Example

In order to illustrate the application of the sensitivity analysis, once again the \( H_\infty \) controller designed by [Tsai (1989)] will be used. When this controller is converted to \( z \)-operator form (the \( c2dm \) function in MATLAB will do this), the resulting controller gives the following sensitivity factor matrices for the \( A_z \), \( B_z \) and \( C_z \) matrices (given to 2 decimal places):

\[
S_{ZA} = \begin{bmatrix}
3631.83 & 0.11 & 4.20 & 0.08 & 0.00 & 0.00 & 0.00 \\
0.32 & 67.02 & 0.33 & 0.05 & 0.01 & 0.00 & 0.00 \\
4.02 & 0.59 & 897.31 & 0.03 & 0.02 & 0.00 & 0.00 \\
0.09 & 0.01 & 0.07 & 14.95 & 0.01 & 0.00 & 0.00 \\
0.03 & 0.00 & 0.01 & 0.02 & 2.35 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.19 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.08
\end{bmatrix}
\]

\[
S_{ZB} = \begin{bmatrix}
15.96 & 0.92 & 1.25 & 0.06 & 0.02 & 0.00 & 0.00 \\
15.55 & 0.30 & 186 & 0.07 & 0.00 & 0.00 & 0.00
\end{bmatrix}
\]
The sensitivity factor matrices for the $A_{\delta mc}$, $B_{\delta mc}$ and $C_{\delta mc}$ matrices (Note that the subscript $\delta mc$ is used to indicate that the system is in modified canonic delta form) are:

$$
S_{\delta mc} = 
\begin{bmatrix}
0.2580 & 0.4737 & 0.7024 & 0.8132 & 0.5070 & 0.3764 \\
0.2265 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.3463 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.4049 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.3384 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.1886 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.0576
\end{bmatrix}
$$

$$
S_{\delta mc} = [0.4505 \ 0 \ 0 \ 0 \ 0 \ 0]^T
$$

$$
S_{\delta mc} = [0.1048 \ 0.2441 \ 0.4150 \ 0.5585 \ 0.5673 \ 0.4461 \ 0.3700]
$$

The sensitivity factors in the above matrices are normalised and can be interpreted as [Forsythe and Goodall (1991)]:

- Sensitivity factor $\approx 1$ ('Normal' sensitivity)
- Sensitivity factor $>> 1$ ('High' sensitivity)
- Sensitivity factor $<< 1$ ('Low' sensitivity)

As can be seen from the results above, the $\delta$-operator modified canonic system has 'normal' or 'low' sensitivity to changes in the coefficients, whereas the $z$-operator system has 'high' sensitivity to changes in a significant number of the coefficients. The sensitivity factors of the $A_z$ matrix are particularly large, when compared to the sensitivity factors of the $A_{\delta mc}$, and this means that great care has to be taken in implementing the coefficients of the $z$-operator system. A greater number of bits need to be provided for the $z$-operator system, than for the $\delta$-operator system, to achieve a performance which is acceptable. If an adequate number of bits are not provided for the coefficients then it is possible that the quantised system will be unstable, although its performance will be unacceptable long before instability occurs.

Simulations were also carried out to compare the step responses of the original system to the responses of the system when the coefficients were changed by a certain percentage. Figure 4.2 shows the response of the systems when the
coefficients of the $A_z$ matrix in the $z$-operator formulation are increased by 0.1% and when the coefficients of $A_{\delta\nu\epsilon}$ matrix of the $\delta$-operator system are increased by 5%.

![Plot](image)

**Figure 4.2** - Comparison of sensitivity of $z$-operator formulation and $\delta$-operator formulated modified canonic system

In general it has been found that the $z$-operator systems are extremely sensitive to changes in coefficients and the resulting sensitivity factors particularly large. The $\delta$-operator systems have been found to have 'normal' sensitivity to changes in coefficients and the sensitivity factors are normally around 1. The above results are clear in Figure 4.2 where it can be seen that the error due to a 0.1% change in the $z$-operator formulation is far greater than the error due to a 5% change in the $\delta$-operator formulation. This is reflected in the sensitivity factors calculated earlier in the section.

Note that the $A_z$, $B_z$, $C_z$, $A_{\delta\nu\epsilon}$, $B_{\delta\nu\epsilon}$ and $C_{\delta\nu\epsilon}$ matrices are given in Appendix G. Also included in the Appendix is the MATLAB function written to calculate the
The sensitivity factors produced above can be used to determine the wordlengths of the coefficients using similar techniques to those employed by [Forsythe and Goodall (1991), pp 124 onwards]. That is, use the sensitivity factors to determine the accuracy required in the coefficients to achieve a certain accuracy in the magnitude of the frequency response. This coefficient accuracy requirement can then be directly used to calculate the number of bits required for the coefficients. Further detail can be found in the above reference.

4.8 Summary

Two alternative methods that can be used to obtain the modified canonic state space structure have been presented. The first method, as described in section 4.5.1, is straightforward but assumes that the original discrete z-operator based system is represented in controller canonical form. It is prone to ill conditioned matrices and numerical problems can give a system which is 'wrong'. The second method, described in section 4.6, is not as straightforward but is definitely more robust and can easily be coded in software easily.

To test the robustness of the structures a transfer function magnitude sensitivity approach was outlined. This approach is a highly practical method to use in determining the coefficient wordlength requirements for a state space system. The modified canonic structure is, as expected, highly robust to changes in coefficients when compared to the z-operator structure. To compare the robustness, the step responses of the systems when the coefficients are changed by a certain percentage were examined and this confirmed the results of the sensitivity analysis.
Chapter 5

A CONTROL SYSTEM PROCESSOR (CSP)

The previous two chapters looked at structures which can be used to implement controllers that are specified in either transfer function format or state space format. In both cases the structures are based on the $\delta$-operator and were shown to be robust to changes in coefficients, hence requiring fewer significant bits to represent the coefficients of the $\delta$-operator based structure as compared to the $z$-operator based structure. Additionally, the structures also yield an internal variable wordlength (total wordlength being the sum of basic plus overflow plus underflow as seen in Figure 3.1) that is less than that produced by an equivalent $z$-operator based structure.

The next stage is to translate these wordlength requirements into a specific hardware and software combination which can be used to implement a controller in an accurate, effective and efficient manner. Chapter 2 showed that, for digital implementation, the controller can be implemented on a number of different hardware platforms. Of all these platforms only the 'general purpose control system processor' concept is considered.

In order to do this the chapter starts by considering how the CSP fits into the control field and how the speed of the processor can be improved. This will be followed by a discussion on the coefficients and variables of the controllers. A definition of the format which will be used to represent the coefficients will be presented. The internal variable format is then briefly discussed. This is followed by an examination of the typical operations the processor will have to perform and from this an instruction set for the processor can be derived. Finally, the core construct of the processor will be discussed and this will include sections on registers,
adders/subtractors and a discussion on the design of a multiplier.

5.1 The CSP in the Control Field

In a simple digital control system the processor implements a control law which ensures that the plant is adequately controlled. The control law is effectively an algorithm in software which processes some measured signal(s) and then outputs the result(s) to the real world. This procedure is repeated every sample period. Usually an Analogue to Digital Converter (ADC) with a ZOH built in is used to convert the measured signals from a transducer into digital form and a Digital to Analogue Converter (DAC) is used to perform the reverse operation when results are output. Also present in the digital controller are EPROM's (Read Only Memory) to hold the program (and controller coefficients) and some Random Access Memory (RAM) to store temporary variables/results. A block diagram of a generalised digital controller architecture is shown in Figure 5.1.

From the block diagram it can be seen that the CSP needs to get data from ADCs, RAM and EPROM. It also writes data to DACs and RAM. Additionally, the CSP processes the incoming signals to produce control outputs. For simplicity it will be assumed that the processor will be responsible only for the implementation of the control algorithm; that is the processor receives an interrupt from the ADC signaling that the new input is ready to be processed. It will be assumed that other housekeeping tasks, such as initiating an analogue to digital conversion, will be performed by off chip timers which can be initialised such that on reset they assume a preset cycle of operation.
An important aspect which will need to be considered is how to speed up the processor? One way in which the processing speed can be improved is by reducing the number of off-chip accesses that the processor has to make. This can be done by providing enough on-chip registers that the processor can use to store internal variables and coefficients. Another method of increasing the speed is to ensure that the instructions are decoded in hardware (hardwired instructions) rather than software (microprogrammed). This is only feasible if the number of instructions is kept to an absolute minimum otherwise the control unit of the processor, where the instructions are decoded, becomes complex and ends taking up too much silicon space. Yet another way in which the processor efficiency can be improved is by introducing pipelining into the design. This, however, is undesirable for real time operation as the pipeline needs to be cleared when an interrupt occurs. This introduces latency. For simplicity it was decided that pipelining will not be considered in the design. These are all examples of applying a RISC [Hennessy and Patterson (1990)] approach to processor design. The processing speed can also be improved by providing a hardware multiplier followed by a barrel shifter; the shifting would take place after the multiplication, which is especially useful when
the result of the multiplication needs to be shifted. This, as will be seen later, is the case for control law multiplications.

Another design consideration is whether to use the Harvard architecture (as used in DSP's), which has separate buses for program and data, or use the Von Neuman architecture. The Harvard architecture was preferred because the bottlenecks associated with Von Neuman architecture could be avoided.

5.2 Coefficients and Variables

As the order of the controller increases, the number of coefficients and the number of internal variables of the controller increases. These numbers will determine the amount of on-chip storage that must be provided for in the CSP. To understand the numbers involved, consider the variation of the numbers as the order of the controller increases. Careful examination of different orders of controllers reveals numbers given in Table 5.1. It represents the number of internal variables and the number of coefficients (assuming that controller will be specified as a modified canonic structure) that are needed for each order of controller.

<table>
<thead>
<tr>
<th>Order</th>
<th>Internal Variables</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>18</td>
</tr>
</tbody>
</table>

*Table 5.1 - Controller order vs Internal Variables and Coefficients*

For the table it has been assumed that any higher order controllers (order > 2) will be implemented as a combination of first and/or second order sections. That is, a
7th order controller will be made up of three second order controllers in cascade with a first order controller. This is due to the advantages that are to be gained in the wordlengths of the internal variables.

The next stage is to consider the formats of the coefficients and the variables. In the past the research has tended to concentrate on specific issues which negate the generality. In the discussion that follows, the emphasis has been to use a semi-analytical approach such that a generalised wordlength and format can be derived. The results that are produced are correct in the sense of order of magnitude and it was felt that this was adequate to demonstrate the concept of a generalised CSP.

5.2.1 Coefficient Format

To consider the coefficient format first the following points need to be appreciated:

- In a control system one nearly always needs gains > 1 by the time they have been scaled - this is related to the concept of proportional band (Note that the concept of proportional band is normally only applied to Proportional type of controller and a typical value is around 5%. After scaling and feedback this relates to a control system gain of about 20). This means that such gains must be accommodated, with saturation limiting. The above argument also holds for analogue implementations as well, but the limiting happens naturally.

- From a filtering point of view it is better to separate out the problem by converting the filters such that they all have an overall gain ≤ 1, otherwise a lot of time would be spent in overflow checking. This implies that any internal variable overflow must be accommodated without saturation, otherwise the digital implementation will degrade the performance compared to the analogue implementation.

Hence for the approach of the work in the thesis, the following arguments are made for ignoring gains > 1 and overflow handling on the processor:
The main emphasis of the research is upon the structure and operation of the recursive computations, and the corresponding processor architecture. It is recognised that left shift/gains > 1 (with limiting) are necessary as a separate facility, but since these are standard procedures they have not been accommodated within the processor study. Only gains < unity are considered from here on, that is, it will be assumed that the coefficients are normally fractional.

The coefficient is assumed to be unsigned because the sign can always be implemented in software by an addition or subtraction. For a fractional coefficient it is usual to have a number of significant bits and a number of leading zeros, particularly for the \( d_1 \) and \( d_2 \) coefficients. That is, the coefficient will be of the form

\[
\text{coefficient} = 0.000\ldots0XXXX
\]

where XXXX are the significant bits.

To determine the number of significant bits that need to be allowed, consider the following: Since all coefficients are assumed to be fractional, which is true for the \( \delta \)-operator approach once the assumption of gains < 1 has been made, the fractional coefficient sensitivity is always unity or less, that is:

\[
\text{implementation accuracy} \Rightarrow \text{coefficient accuracy}
\]

So if 6 bits are allowed for the coefficient's significant part, this gives:

\[
\text{6 significant bits} \Rightarrow 1 \text{ part in 64 which gives better than 1\% accuracy using rounded values.}
\]

Since an accuracy of 1\% is adequate for most applications (bearing in mind that with analogue implementations the accuracy of the passive and active components is not much better), it was decided to allow for 6 significant bits.

For the size of the number of leading zeros, consider the following: Analysis of the modified canonic and cross coupled structures shows that the 'recursive' coefficients (e.g. \( d_1 \) and \( d_2 \)) are of the order \( \frac{T_s}{\tau_c} \), where \( \tau_c \) is the largest controller time constant.

If a sampling frequency of up to 1000 times the controller frequencies is assumed -- a conservative assumption - then coefficients need to be as small as \( \sqrt{1000} \). This
implies that there will be 10 leading zeros (assuming 1's in the significant bits) to give $\sqrt{1024}$. The $p$, $q$, $r$ coefficients of the modified canonic structure will be significantly larger than the $d$, and $d'$, coefficients. This gives a wide range of coefficients that have to be catered for. Hence the following is proposed for coefficient format.

The coefficient can be implemented in a floating point type of format and the multiplication, with an internal variable, performed by first multiplying the significant bits (that is the mantissa) of the coefficient with the variable and then shifting the result to the right by the number of leading zeroes plus the number of significant bits (effectively the exponent). Hence, the following 10-bit format is proposed for the coefficient:

The most significant four bits will represent in binary form the number of leading zeros (the value of four is chosen because it is estimated that 10 leading zeroes will need to be catered for) and the least significant six bits represent the significant bits of the coefficient. For example if the coefficient is equal to 0.00000001101, then in the 10-bit format proposed above this becomes

$$\text{coefficient}_{10} = 1000\ 110100$$

Note that there are 8 leading zeroes hence the four most significant bits of the coefficient are 1000 (8 in binary) and the other six bits are the significant bits of the coefficient, including two trailing zeros (not shown in the coefficient itself). So for multiplication only 110100 would be multiplied with the variable and the result of the multiplication would be shifted right by 14 places (8 + 6). Similarly, 0.0000010001 becomes 0101100010 in the 10-bit format proposed (and the result of multiplication between 100010 and the variable will be shifted right by 11 places (5 + 6). So under the format, the coefficient can have up to 15 leading zeros and a maximum of 6 significant bits, giving it a total wordlength of 21 bits.
With the above proposal, the smallest number that can be implemented is:

\[
\text{smallest number} = 0.000 \, 000 \, 000 \, 000 \, 000 \, 000 \, 001, = 1/2^{22} = 2.38419 \times 10^{-9} \text{ in base } 10
\]

15 leading zeroes 6 significant bits

while the largest that can be implemented is:

\[
\text{largest number} = 0.111 \, 111, = 0.984375 \text{ in base } 10.
\]

6 significant bits Note: No leading zeroes

This gives a wide range of numbers that can be implemented. It should be remembered that the above is both approximate and conservative to ensure the generality of the CSP.

5.2.2 Variable Format

When considering the variable format, one needs to consider the wordlength to which variables should be represented, and of this wordlength how many bits should be allocated for the basic input/output, and how many for underflow and overflow (as depicted in Figure 3.1).

Consider first the overflow issue. In general overflow provision is only required for any effects internal to the structure. Since it has been assumed that only \(\delta\)-operator modified canonic structures are going to be used, the overflow can be ignored as it is not really an issue that has much influence (however, if the structure to be used was other then of the modified canonic type, then the overflow assumption made would not hold). So the choice that needs to be made is on the basic input/output and underflow wordlength.

In most cases an input/output resolution of 12 bits is usually adequate. Note that the number of input/output bits is not really an issue for the work covered in this thesis as it doesn’t affect the approach. It is really, as was mentioned in Chapter 3, a system issue and the result of the thesis would still apply if the number of bits were different. The figure of 12 was chosen as it was a good working figure and would
be adequate to demonstrate the concept of a general CSP.

The underflow issue requires further consideration. For δ-operator structures, underflow approximately obeys the following relationship:

\[
\text{fractional accuracy} \times \text{coefficient size} \\
= 0.2 \times 0.001 = \frac{1}{5000}
\]

Hence roughly 12 bits

In the above, the fractional accuracy is the accuracy required in the internal variables in response to small inputs; that is, a least significant bit change in the input. Note, the 20% figure is fairly generous. Strictly speaking, with the availability of 4 exponent bits in the coefficients, hence permitting 15 leading zeroes, this is probably not enough to accommodate all coefficient sizes. In the work of [Goodall and Donoghue (1993)] they used a 48 bit wordlength for the variables (including input/output bits) but the ratio of filter to sampling frequency was around 20000 while in the above underflow discussions a ratio of 1000 has been assumed.

So for CSP generality it is assumed that 12 bits of underflow are enough, thereby giving a 24-bit wordlength. At this stage it is worth noting that in general a 24-bit wordlength for the internal variables is good enough for most cases in control but to be safer one could have gone for 32-bits. Wordlengths greater than 32-bits are rarely needed and it is only in the most demanding applications that one would require them. For the purpose of this research it was felt that if the concept could be proved with one wordlength then it could easily be applied to another. The next question that springs to mind is: "How does this relate to conventional microprocessors?"

Conventional fixed point microprocessors are usually 8-, 16- or 32-bits models, and the most commonly used in control applications are usually the 8- or 16-bit models. With these latter choices, the computations for the internal variables have to
performed by writing routines to carry out the required arithmetic operations on multiple processor words which are needed to represent the variables, and the calculations therefore take a number of instruction cycles. By contrast, the choice of an appropriate wordlength for the CSP will lead to the calculations being performed in a single instruction cycle.

For the CSP, the two's complement format is used as it is easier to perform additions and subtractions on signed numbers. However, the multiplication between a two's complement number and an unsigned one is slightly more complicated because the sign is embedded in the number. The multiplication procedure will be described in some detail in the section 5.4.3.

53 The Instruction Set

Now that the coefficients and variables have been considered, the design of the instruction set can be examined. To derive the instruction set, recall that initially the processor will only be responsible for implementing the control law. So upon start up, the controller will set up, with the aid of a program stored in an EPROM, and await the timing signal. Upon receiving the timing signal, the inputs will be processed and the control actions output; it will then await another timing signal and the above process will be repeated. This can be summarised as shown in the flowchart of Figure 5.2.

From the flowchart the following groups of instructions are needed:

- for storing the coefficients in a coefficient register (C-Reg) file (the word 'file' is used to denote a collection of registers),
- for implementing the control law, ie read input, process and output action
5.3.1 Instructions

For storing the coefficients in the C-Reg file, it was decided that the coefficients will be moved to the accumulator first using the load immediate instruction, \textit{ldi}, and then the store coefficient instruction, \textit{stc}, will be used to place the accumulator value into the C-Reg file. Other instructions that will be needed will be the jump to an address instruction, \textit{jmp}, which will service interrupts and handle other looping functions when awaiting interrupts. When the processor is interrupted, it will be assumed that the location of the interrupt service routine (ISR) will be at address 02H.

The remaining instructions are for implementing the control law. In order to derive what these instructions might be, the equations needed to implement the structures...
of chapters 3 and 4 were examined. For example, consider the second order modified canonic structure of Figure 3.2. The equations to implement the structure are

\[
\begin{align*}
  v &= u - w - x \\
  y &= p_1 v + p_2 w + p_3 x \\
  x &= x + d_1 w \\
  w &= w + d_1 v
\end{align*}
\]

By writing this in pseudocode (see Appendix H), the following key instructions and their descriptions can be determined - see table 5.2.

5.3.2 Encoding Format

The first question is what is the width of the instructions? After careful inspection of the requirements and the instructions it was found that a width of 16 bits will be enough. The 16 bits includes provision for the sizes that will be chosen for the register files; this will be discussed in section 5.4.1 when the register issue is considered in further detail. Of the 16 bits, the opcode of the instructions will occupy 5 bits, as the total number of instructions for the prototype are 16 and the remaining 11 bits can be configured as appropriate for each instruction. The encoding formats shown in Figures 5.3a and 5.3b were found to be adequate for the instructions presented in section 5.3.1.

![Figure 5.3a - Encoding format 1](image)

![Figure 5.3b - Encoding format 2](image)
A Control System Processor

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>nop</em></td>
<td>no operation</td>
<td>Dummy instruction to do nothing. PC not changed.</td>
</tr>
<tr>
<td><em>sub</em></td>
<td>subtract</td>
<td>Subtract value in V-Reg file from value in Accumulator. Result in Accumulator.</td>
</tr>
<tr>
<td><em>add</em></td>
<td>add</td>
<td>Add value in V-Reg file to value in Accumulator. Result in Accumulator.</td>
</tr>
<tr>
<td><em>mac</em></td>
<td>multiply and</td>
<td>Multiply coefficient with variable and add result to value in Accumulator. Result in Accumulator.</td>
</tr>
<tr>
<td></td>
<td>accumulate</td>
<td></td>
</tr>
<tr>
<td><em>ldv</em></td>
<td>load variable</td>
<td>Load value from V-Reg file to Accumulator.</td>
</tr>
<tr>
<td><em>ldm</em></td>
<td>load memory</td>
<td>Load value from memory location into Accumulator.</td>
</tr>
<tr>
<td><em>ldi</em></td>
<td>load immediate</td>
<td>Load immediate value into Accumulator.</td>
</tr>
<tr>
<td><em>stv</em></td>
<td>store variable</td>
<td>Store value in Accumulator to V-Reg file.</td>
</tr>
<tr>
<td><em>stm</em></td>
<td>store memory</td>
<td>Store value in Accumulator to memory location.</td>
</tr>
<tr>
<td><em>stc</em></td>
<td>store coefficient</td>
<td>Store value in Accumulator to C-Reg file.</td>
</tr>
<tr>
<td><em>clr</em></td>
<td>clear</td>
<td>Clear Accumulator to zero.</td>
</tr>
<tr>
<td><em>jmp</em></td>
<td>jump</td>
<td>Jump to memory location specified.</td>
</tr>
<tr>
<td><em>rti</em></td>
<td>return from</td>
<td>Return processor to pre-interrupt state.</td>
</tr>
<tr>
<td></td>
<td>interrupt</td>
<td></td>
</tr>
<tr>
<td><em>rad</em></td>
<td>read from ADC</td>
<td>Read value from ADC into Accumulator.</td>
</tr>
<tr>
<td><em>wda</em></td>
<td>write to DAC</td>
<td>Write Accumulator value to DAC.</td>
</tr>
<tr>
<td><em>eni</em></td>
<td>enable interrupt</td>
<td>Switch to determine whether processor should respond to an interrupt.</td>
</tr>
</tbody>
</table>

Table 5.2 - CSP Instructions and their descriptions

The encoding format of Figure 5.3a will be used by the instructions *rti, clr, nop, jmp, sim, ldm, rad, wda* and *ldi*. Obviously for some of the instructions the contents of the address/immediate value field will be irrelevant. On the other hand, the encoding format of Figure 5.3b will be used by the instructions *mac, stv, ldv, sub, add and stc*. Once again, in some cases the appropriate field contents will be used while the remaining field contents will not be used.
5.4 The Core of the Processor

Initially the processor’s main aim is to complete the calculations as fast as it can. In order for this to be achieved a number of points were mentioned in section 5.1. The core of the processor will be determined by how the registers and the arithmetic unit are organised. The registers have to provide storage for both the coefficients and the internal variables (or temporary results), while the arithmetic operations that need to be performed are additions, subtractions and multiplications.

5.4.1 The Registers

If on-chip storage, in the form of registers, is provided for the coefficients and internal variables then a reduction in the processing time can be achieved. According to the formats described in sections 5.2.1 and 5.2.2, the coefficients and variables will have different wordlengths; it is therefore better to provide two different register files for the coefficients and the variables. From the discussion above it can be seen that the width of the coefficient register file (C-Reg) will be 10 bits while the variable register file (V-Reg) will have a width of 24 bits.

The other decision that has to be made is with regard to the size of the two register files. That is: How many registers should be provided in each of the files? At this stage, it should be observed that the number of registers that can be provided for is limited by the 16-bit width that has been chosen for the instructions. Since the opcodes take up 5 bits, only 11 bits are available for the addressing of the registers. Of these 11 bits, 4 of the bits will be used to address the V-Reg and 5 bits to address the C-Reg file. If any further address bits are necessary as a result of increased size of the register files, then it will be necessary to reconsider the encoding format proposed in section 5.3.2.

With the encoding format structured such that provision is made for 32 coefficients and 16 variables, then controllers up to 7th order can be implemented such that all the necessary storage will be available on-chip and there will therefore be no need
to store any intermediate variables off-chip in memory. The analysis of section 5.2 will confirm this.

Other registers that are necessary are those which will be needed by the processor to function appropriately. These are

- a Program Counter (PC) which will contain the address of the next instruction to be executed; after each instruction word is fetched, the PC will be incremented by one to point to the next word.

- an Instruction Register (IR) which will store the current instruction that is being serviced

- an Interrupt Address Register (IAR) to store the address giving the location of the ISR (Interrupt Service Routine)

- a Return from Interrupt Address Register (RIA) where the address of the current instruction which the processor was executing when interrupted will be stored; when an interrupt occurs, the PC value is stored in RIA, the PC is loaded with the address of the ISR (which is obtained from the IAR), the interrupt handling routine is executed and on completion (return from interrupt) the PC is loaded with the address stored in RIA

- a Product Register

- an Accumulator (ACC)

5.4.2 The Adder/Subtractor Unit (ASU)

The ASU will add or subtract two 24-bit two's complement numbers. The additions and subtractions are necessary arithmetic operations as can be seen from the equations of section 5.3.1. At the heart of the ASU is a 24-bit Carry Look Ahead
A Control System Processor

(CLAs) Adders. Various different adder configurations ranging from a ripple carry (RC) adder to a combination of RC and CLA to a CLA adder are possible. The CLA adder was chosen because of the speed at which the calculations can be accomplished. The speed improvement is especially noticeable when the number of bits that needs to be added is large. A block diagram of the ASU is given in Figure 5.4.

![Block diagram of ASU](image)

**Figure 5.4 - Block diagram of ASU**

5.4.3 The Multiplier

The multiplier needs to be able to multiply a two’s complement number (the variable) with an unsigned one (the coefficient). The process is not easy to understand and is best illustrated with the aid of an example. Let us look at the problem of multiplying $1543_{10}$ (represented as a 24-bit two’s complement number) by $55_{10}$ (represented as an unsigned number). This is a positive by positive multiplication and is straightforward as shown in Figure 5.5.
Now consider the problem of multiplying a negative number with an unsigned number. Multiplying \(-154_{10} (= 1111 1111 1111 1001 1111 1001)\) and \(55_{10}\) by long hand as in Figure 5.5, gives 11011011111101011010001001111, which is incorrect. The correct answer should be 11 1111 1111 1111 0100 0111 1111, (= -84865,\(_{10}\)). It is obvious that the sign needs to be taken into account during the multiplication procedure. Therefore, the algorithm needs to be examined in further detail.

**The Algorithm**

The algorithm described below is an adapted version of "A Two's Complement Parallel Multiplication Algorithm" described in [Baugh and Wooley]. Baugh and Wooley described a procedure to follow for multiplying 2 two's complement numbers together. This procedure is modified slightly for the case of multiplying a two's complement number with an unsigned number and is presented below.

As has already been noted, the difficulty in the multiplication of the variable and coefficient lies with the sign of the two's complement variable. To analyse the procedure in detail, let \(V_v\) be the value of the variable and \(C_v\) be the value of the coefficient. The n-bit variable in two's complement representation is given by

\[ V_v = -V_{12} \cdot 2^{n-1} + \sum_{i=0}^{n-2} V_i \cdot 2^i \]  \hspace{1cm} (5.1)

and the m-bit coefficient in unsigned representation is given by

\[ C_v = \sum_{j=0}^{m-1} C_j \cdot 2^j \]  \hspace{1cm} (5.2)

Figure 5.5 - Multiplication of 2's complement number with unsigned number
Multiplying $V_n$ and $C_j$ gives a product $P$ whose value $P_v$ is

$$P_v = \left[ -V_n \cdot 2^{n-1} + \sum_{j=0}^{n-2} V_j \cdot 2^j \right] \left[ \sum_{j=0}^{n-2} C_j \cdot 2^j \right]$$

$$P_v = -\sum_{j=0}^{n-2} V_n \cdot C_j \cdot 2^{n-1+j} + \sum_{j=0}^{n-2} V_j \cdot C_j \cdot 2^{j}$$  \hspace{1cm} (5.3)

When forming $P$ by adding the partial products, the sign of bit $V_n \cdot C_j$ for $j = 0, 1, ..., m-1$ in each partial product must be considered as it is negative, that is the left most bit in each row of partial products shown in Figure 5.6.

**Figure 5.6 - Conventional n-bit by m-bit multiplication for m = 6**

One way of forming the product is to omit the $V_n \cdot C_j$ bit from each of the partial products and adding them first to form an intermediate result. The negative bits of the partial products can then be subtracted from this intermediate result to form the product as shown in Figure 5.7.

**Figure 5.7 - Multiplication with Correction for sign**
Alternatively, instead of subtracting, the negation of these partial product bits can be added to the intermediate result. The negation of a two's complement number $z = (z_{k-1}, ..., z_0)$ with a value $z_i$ is

$$z_v = 1 - z_{i-1}2^{k-1} + \sum_{j=0}^{k-2} z_j 2^j$$

(5.4)

where $z_{i-1}$ is the complement of $z_i$. Therefore, the subtraction of

$$\sum_{j=0}^{m-1} V_{n,i} C_j 2^{n-1-j} = 2^{n-1} \left(-0.2^m + \sum_{j=0}^{m-1} V_{n,i} C_j 2^j \right)$$

(5.5)

can be replaced by the addition of

$$2^{n-1} \left(-1.2^m + 1 + \sum_{j=0}^{m-1} \overline{V}_{n,i} \overline{C}_j 2^j \right)$$

(5.6)

that is, the partial product row

0 $C_3 V_{n-1}$ $C_4 V_{n-1}$ ... $C_0 V_{n-1}$

is replaced by

1 $\overline{C}_3 \overline{V}_{n-1}$ $\overline{C}_4 \overline{V}_{n-1}$ ... $\overline{C}_0 \overline{V}_{n-1}$

and a "1" is added to the $P_{n,i}$ column of the intermediate result.

All the partial product bits can now be treated in exactly the same manner with respect to sign but some of the partial product bits are NAND while others (most) are AND.

To simplify and keep it uniform, it should be noted that equation (5.6) has the value

$$2^{n-1}(-2^m + 1 + \sum_{j=0}^{m-1} \overline{C}_j 2^j) \text{ for } V_{n,i} = 0$$

$$2^{n-1}(-2^m + 1 + \sum_{j=0}^{m-1} \overline{C}_j 2^j) \text{ for } V_{n,i} = 1$$

(5.7)

and can therefore be written as

$$2^{n-1} \left(-2^m + \overline{V}_{n,i} 2^m + V_{n,i} + \sum_{j=0}^{m-1} V_{n,i} \overline{C}_j 2^j \right)$$

(5.8)

The above can be interpreted as:
- add 1 at position n-1+m (1st term in the brackets)
- add $V_{n+1}$ at position n-1+m (2nd term in the brackets)
- add $V_{n+1}$ at position n-1 (3rd term in the brackets)

Therefore, the complete algorithm for multiplying a two’s complement variable with an unsigned number can be given as in Figure 5.8 for the case of m=6.

![Figure 5.8 - Final Algorithm for Multiplying a 2’s Complement number with 6-bit unsigned number](image)

Once the partial products have been generated using this algorithm, the next step is to add the partial products quickly. A number of different policies exist and are outlined in [Ma and Taylor (1990)]. The Wallace tree approach was preferred to reduce the number of partial products from 6 to 2. The final two products can then be fed to carry look ahead adder to get the final product which then needs to be shifted right by the appropriate number of places (the sum of the number of leading zeros and the number of significant bits) to obtain the correct result between the multiplication of a coefficient and a variable.

### 5.4.4 A Generic Architecture

Based on the discussions in the previous sections, a block diagram based schematic for the processor can now be derived. As has been seen, the core will be how the registers and arithmetic unit interact within the processor. Some of the essential features of the processor are
• The reduced instruction set, thereby making it possible for the instruction decoding to be hard wired (quicker execution) rather than microprogrammed.

• The asymmetrical multiplier, which will perform a 24 by 6 bit multiplication of a 2's complement variable with an unsigned coefficient. The product will then be scaled by a (barrel) shifter to implement a simple exponent.

• The register files, which are available for storage of the internal variables and coefficients, thereby reducing the number of reads and writes that need to be performed to external memory.

• The CSP accesses data as 24 bit words and this is usually stored in data memory (RAM); instructions for the CSP are 16 bit words and are stored in instruction memory (EPROM); since the instruction size is 16-bit, of which the opcode takes up 5 bits, an address of upto a maximum of 2K memory locations ($2^{11}$) is available.

• Other features that should be provided for are on-chip timers and counters. These can be used to provide the timing functions such as an internal interrupt facility and also make it possible to initiate and synchronise other off chip tasks such as analogue to digital conversion. Such facilities are an essential part of a control system.

The above features are included in the architecture defined for a generic CSP as can be seen in Figure 5.9. The schematic is not a complete design and only includes the key functions, in block format, that are necessary for the CSP to operate.

This schematic will be used as a guide when a behavioural model for the CSP is developed in VHDL, which is discussed in detail in chapter 6.
Figure 5.9 - A Generic Architecture for a CSP
Summary

An idea for a prototype CSP has been presented and some of the essential features, such as:

- coefficient and internal variable formats,
- the instructions and their encoding formats

have been discussed. The architecture of the processor was also discussed and the register issue and the multiplication procedure, that will result due to the two different formats, were considered. Finally an architecture for a CSP with the essential features included was presented.
In Chapter 5 a basic instruction set for the CSP was presented and other issues such as coefficient and variable formats discussed. Also presented were the key features that the CSP needs to perform the required operations:

- general purpose registers (the C-Reg and V-Reg files);
- software addressable registers;
- specific purpose registers (e.g. the PC, IAR, RIA) which cannot be accessed by the user;
- and the arithmetic unit, (i.e. the adder and multiplier plus shifter).

In this chapter a bus architecture for the CSP will be described first, and by combining this with the discussions of Chapter 5, a behavioural description of the processor will be presented. To test the behavioural description, a test bench model is constructed which will be checked by implementing a phase advance controller. The results of the behavioural model to a step input will be compared with the results of a simulation of the phase advance controller using MATLAB. Finally, the issue of synthesis will be discussed. Synthesis is a method whereby the structure of a circuit can be inferred from its VHDL description.
6.1 The Bus Architecture

6.1.1 The CSP Port

The CSP communicates with the external peripherals, such as ADC, DAC, RAM and EPROM, over a synchronous 11-bit address bus and a 24-bit data bus. Along with the communication ports a number of other ports are also present which provide the necessary control signals that the CSP needs for synchronous communication. The external ports of the processor are as shown in Figure 6.1.

![Diagram of the CSP Port]

**Figure 6.1 - The CSP Port**

The two clock inputs, $\phi_1$ and $\phi_2$, provide a two-phase non-overlapping clock for the processor. The waveforms for the clock are shown in Figure 6.2. Each cycle of the $\phi_1$ clock defines a bus state, one of $T_i$ (idle), $T_1$ or $T_2$. Bus transactions consist of a $T_1$ state followed by one or more $T_2$ states, with $T_i$ states between transactions. More details will be given when the timing of the bus transactions is discussed (section 6.1.2 onwards).
The A_BUS is a 11-bit address bus, and the D_BUS is a 24-bit bi-directional data bus. The RD_BAR and WR_BAR ports control the bus read and write transactions. The DATA_BAR port is a control signal to indicate that the read/write being performed is from/to a RAM while PROG_BAR port is a control signal indicating that the bus read in progress is an instruction fetch (usually from an EPROM) and the IOS_BAR port is a control signal to indicate the bus is communicating with either the ADC subsystem or the DAC subsystem. The READY_BAR input is used by the peripherals to indicate that the data requested by the processor is ready or the write data has been accepted. The INT_BAR input is used by the ADC subsystem to interrupt the processor when it has completed the conversion process and the data is available for the processor to read. Note that for simplicity, it has been assumed that the start conversion process will be initiated by off-chip timers. The RST_BAR port is used to reset the processor to an initial state.

6.1.2 Memory Read

The timing for the memory read transaction is shown in Figure 6.3. During an idle state, T0, the processor places the memory address on the address bus to initiate the transaction. Note that the processor uses the leading edge of the \( \phi_2 \) clock to synchronise this event. In the next state, T1, after the leading edge of \( \phi_1 \) clock, the processor asserts the read control signal, indicating that the address is valid and the memory should start the read transaction. Additionally, the processor also asserts the either the PROG_BAR signal, if it is reading instructions, or the DATA_BAR signal.
signal, if it wants data from the RAM. The remaining control signals are left inactive during the memory read transaction.

During the T1 and the following T2 state, the memory accesses the data/instructions and places it on the data bus. If it has completed the data access by the end of the T2 state, it asserts the ready active signal. The processor accepts the data and the transaction is complete. If, however, the memory has not yet completed accessing the data, then the ready is left inactive and the processor repeats T2 states until ready active is detected. At the end of the transaction, the processor changes all the control outputs to their default state and the memory deactivates READY_BAR and removes data from the data bus.

6.13 Memory Write

Once again the transaction commences when the processor places an address on the address bus during the Ti state. After the leading edge of $\phi_1$, during the subsequent T1 state, the processor activates the WRITE_BAR and DATA_BAR signals to indicate a memory write. The remaining control signals are left inactive for the duration of the transaction. During the T1 state the processor also places the data to be written on the data bus. The timing diagram is shown in Figure 6.4. Once again
it should be noted that the processor uses the leading edge of the \( \phi_2 \) clock to synchronise the data and address placement on the respective buses.

![Timing Diagram of Memory Write Transaction](image)

**Figure 6.4 - Memory Write transaction**

The memory can accept this data during the T1 state and subsequent T2 states. Like for the memory read transaction, if it has completed the write by the end of the T2 state, it asserts READY_BAR. The processor then completes the transaction and continues with T1 states, and the memory deactivates READY_BAR. Once again, if the memory has not completed the write by the end of the T2 state, it leaves READY_BAR inactive and the processor repeats T2 states until it detects an active READY_BAR signal.

6.1.4 **ADC Read**

The processor initiates the ADC read transaction when it activates the IOS_BAR signal in a T1 state. After the leading edge of \( \phi_2 \) in the following T1 state, it activates RD_BAR signal and the remaining control signals are left inactive. The timing diagram is shown in Figure 6.5.

For simplicity it has been assumed that the ADC subsystem will then place the data on the data bus during the subsequent T2 state and activate the READY_BAR signal to indicate that the data is ready to be accessed by the processor. The processor
reads the data and then deactivates the IOS_BAR and RD_BAR. Once the ADC subsystem detects that the data has been read, it removes the data from the bus and deactivates READY_BAR.

Figure 6.5 - ADC Read transaction

6.15 DAC Write

Once again the transaction begins when the processor activates IOS_BAR during the T1 state. In the T1 state that follows, the WR_BAR control signal is asserted and the remaining control signals are left in their inactive states. The data is also placed on the data bus by the processor during the T1 state. The timing diagram is shown in Figure 6.6.

Figure 6.6 - DAC Write

Once the DAC has read the data during the following T2 states, it asserts READY_BAR and the processor deactivates the IOS_BAR and WR_BAR control signals. The data is then removed from the data bus and READY_BAR deactivated.
6.2 The CSP Model

The behavioural description that will be derived in this section can be used to execute test programs in the CSP instruction set by connecting it to a test bench, the model for which is given in section 6.3. The model is generated in VHDL. VHDL is a result of the United States of America's Department of Defense project initiated to produce an industry standard language for expressing and simulating digital designs. The simulation of a description in VHDL is controlled by the language definition. The results from a VHDL simulation should always be the same regardless of the toolset being used. VHDL is the fastest growing design entry method and its main advantage is its ability to express a design in abstract form meaning that a concept's validity can be verified before one attempts to build it [Perry (1991)]. For these reasons VHDL was chosen as medium through which the concept of the CSP should be verified.

Basically, the model for the CSP should fetch, decode and execute the instructions. In order for the CSP to perform these tasks, a number of procedures are written to mimic some of the operations. For example, the fetch sequence is essentially a memory read transaction and the model should therefore obey the timing of Figure 6.3 (with PROG_BAR taken low instead of DATA_BAR low) and the procedure EPROM_read is written to satisfy the timing. Other procedures that are written are

- *RAM_read* to read from the RAM satisfying the timing diagram of Figure 6.3,
- *RAM_write* to implement the memory write transaction and hence satisfy the timing diagram of Figure 6.4,
- *ADC_read* to read data from the ADC_subsystem (see test bench model description),
- *DAC_write* to write data to the DAC_subsystem,
- *Add* which adds two two's complement numbers,
- *Multiply* to perform the multiplication of a two's complement number with an unsigned number and the result is then right shifted

Architectural Considerations for a CSP

142
appropriately to obtain the correct result as described in section 5.4.3.

Once the instruction has been fetched the program counter (PC) is incremented by one to point to the location of the next instruction. Decoding can now take place and the opcode part of the instruction is examined to deduce which instruction it is. The instruction is executed by either calling the relevant procedure, if either communication over the bus is necessary or the instruction is an arithmetic operation, or by simple VHDL statements which perform the required operation. The complete VHDL listing of the model is given in Appendix I.

6.2.1 Data Bus Connections

The most important point of the model is the way the processor's data bus connections have been routed. It has been seen that the choice of the data bus is 24-bit. Of the 24-bits, 12 are for basic input output (the most significant bits) and 12 are for underflow (the least significant bits) and the decimal point is located in between the two 12-bit sections. So when data is retrieved from the ADC subsystem, the 12-bits of data that are read are aligned such that they become the 12 most significant bits of the accumulator. The least significant 12-bits are made equal to zero. Similarly when data is written to the DAC subsystem, only the 12 most significant bits are necessary and hence only the 12 most significant bits of the accumulator are written. Additionally, note that the instructions are 16-bit wide and therefore the EPROM will be connected to the data bus via the 16 least significant bits of the data bus. These points are reflected in the test bench model that will be constructed. Finally, remember that the aim is to implement modified canonic structures on the processor, for which it has been shown that overflow of the internal variables is not a problem.
6.2.2 Coefficients

The CSP needs to provide a route such that at the start of the control software, the coefficients are first moved to the on-chip C-Reg file. This is accomplished by moving the coefficients into the accumulator using \textit{ldi} (load immediate) instruction and then storing this accumulator value to the appropriate C-Reg address using the \textit{stc} (store coefficient) instruction. When the immediate value is loaded, it is placed into the least significant 10-bits of the accumulator. It is these 10-bits that are eventually routed during a \textit{stc} instruction to the C-Reg address.

6.2.3 Interrupts

When an external interrupt occurs, the value of the PC is saved in the RIA register and the PC is then loaded with the value of the IAR (02H). This initiates the execution of the ISR. At the end of the ISR, the \textit{rti} instruction ensures that the processor is returned to the state it was in before the interrupt (PC is loaded with the value in RIA). Note that for simplicity it has been assumed that the interrupt will be the timing signal from the ADC (see Figure 5.2).

6.2.4 Initial State

On a RESET_BAR active signal, the processor is initialised such that the PC, RIA, MAR and Acc registers are set to 00H. The IAR is set to point to location 02H. Also, the signals on the READ_BAR, WRITE_BAR, IOS\_BAR, DATA\_BAR and PROG\_BAR ports are all asserted high. The D\_BUS is placed in an high impedance (z) state.
6.3 The Test Bench

The test bench for the CSP contains peripherals that are common to a control application. As was seen in Figure 5.1, these are typically an ADC, a DAC, a RAM and an EPROM. In addition to this a clock generator is also necessary. The test bench has been modelled in VHDL and the circuit is shown in Figure 6.7. A brief description of the models for each of the parts is supplied in the next few subsections.

![Figure 6.7 - The Test Bench Model](image)

6.3.1 The Clock Generator

The model for the clock generator generates a two-phase non-overlapping clock with waveforms as shown in Figure 6.2. Additionally, the model also generates a reset signal which will simulate a power on reset operation. Every time the simulation is initiated, the RST_BAR signal is held low for 3 cycles of the $\phi_1$ clock. After this, it is held high until the simulation is restarted.
6.3.2 The ADC Model

The ADC subsystem will be responsible for initiating conversion of the input signal, (i.e. the off-chip timers, in Figure 5.1, will be modelled as part of the ADC subsystem) and interrupting the processor, the rate of which will depend on the sample frequency. The model should allow for the sample frequency to be set. For simplicity it has been assumed that the input is a step.

The model will, therefore, send an interrupt every few seconds (depending on the sample frequency) and hold it low (active low) for 3.5 clock cycles. Once the subsystem is selected (CS_BAR low and RD_BAR low), the data is placed on the data bus and it is held there until the chip is deselected (CS_BAR high or RD_BAR high). The READY_BAR low is also asserted to indicate data is valid and it is also held until the de-selection of the chip.

6.3.3 The DAC Model

The DAC subsystem should ideally convert the digital data and output it in analogue form to an actuator. However, for the test bench, the DAC subsystem is modelled such that the data is stored in an array and written to a data file when the system is selected (CS_BAR low and WR_BAR low). In this way the data can be compared with simulation results produced by MATLAB. A READY_BAR low signal is asserted and held until the system is deselected (CS_BAR low or WR_BAR low). This will satisfy the timing diagram of Figure 6.6.

6.3.4 The RAM Model

The RAM is modelled as an array of size 2 kB and width 24-bit. The model is such that on a chip select, the address is read from the data bus. Depending on whether it is a read or write operation, it performs two different functions. When the RAM is in read mode (RD_BAR low), then the data is retrieved from the appropriate memory location and placed on the data bus. READY_BAR low is asserted to
indicate data is valid and held until the processor asserts a RD_BAR high signal.

If it is a write operation that is required (WR_BAR low), then the data is read from the data bus and written to the appropriate memory location. The READY_BAR is taken low when operation is complete and held until WR_BAR is taken high by the processor.

In both cases the timing diagrams of Figures 6.3 (read) and 6.4 (write) are satisfied.

6.3.5 The EPROM Model

The EPROM like the RAM is modelled as an array of size 2 kB and width 16-bits. The EPROM however can only be read from and therefore the model is similar to the RAM read mode. When the chip is selected (CS_BAR low), the address is read from the address bus, the data is then retrieved from the appropriate memory location and placed on the data bus. The READY_BAR is asserted to indicate data is valid and held until RD_BAR is taken high by the processor.

Also, the machine code of the control software will be stored within the model. This is done by initiating the array such that its values are the software statements of the control law being implemented.

6.4 Implementing a Phase Advance Controller

In order to test the model and determine its correctness, a simple first order phase advance controller given in [Forsythe and Goodall (1991), pp. 157-8] was implemented. The response of the controller to a step input of various sizes was noted and these responses were compared with the results from a MATLAB simulation of the controller to the respective step inputs. The controller expressed in modified canonic form is given by the transfer function
\[ H(\delta) = \frac{p + q \delta^i}{1 + d_i \delta^i} \]  

(6.1)

where \( p = 4.8095 \), \( q = 1 \), \( d_i = 0.09524 \). Note that these are the exact values prior to quantisation into the 6 significant bit coefficient wordlength.

6.4.1 The Coefficients

After performing a sensitivity analysis and aiming to achieve a 5% emulation accuracy, it was found that four significant bits for the coefficients would be adequate to meet the emulation accuracy. Using this result, the coefficients can be expressed to their nearest values, to get

\[
\begin{align*}
    p &= 5 (= 101.0), \\
    q &= 1 (= 1.000), \\
    d_i &= 0.09375 (= 0.0001100)
\end{align*}
\]

In the coefficient format proposed in section 5.2.1, \( d_i \), becomes

\[ d_i = 1100 110000 \quad \text{(will be stored in \( Rc0 \) - register 0 of C-Reg file)} \]

i.e. only 4 of the 6 significant bits provided for have been used.

What about \( p \)? So far it has been assumed that the coefficients will be fractional in nature and will therefore be implemented by expressing them in the format given in section 5.2.1; \( p \), however, is greater than unity. Fortunately in this case it is a whole number and the effect of multiplication with \( p \) can be implemented by a series of additions. One should, however, remember that control gains greater than unity are inevitable and these can be handled by either pre- or post-scaling the controller appropriately with a scale factor. For the phase advance controller one could scale the coefficients by a factor of 8 (divide) and post scale the result by 8 (multiply).

The choice of 8 is made so that the result can be post scaled by 3 left shifts. If any other factor was chosen instead then one would have to multiply the result with that factor which might be more complicated. Therefore, if the factor is a multiple of 2 then the scaling can be achieved easily enough by a number of left shifts.
A Model for the CSP

Finally note that the multiplication with \( q \) will be implemented simply as an addition because in this particular example \( q = 1 \).

6.4.2 The Variables

After performing a steady state analysis on the internal variables, the calculations showed 8 underflow bits will be needed. Hence the total wordlength for the internal variables will be 20 (based on an input/output wordlength of 12 bits). For the example above the two internal variables \((v\) and \(w\)) will be stored in locations \(Rv0\) and \(Rv1\) respectively (registers 0 and 1 in the V-Reg file). Note that although only 20-bits are required for the internal variables, the design of the CSP provides a 24-bit wordlength for the variables.

6.4.3 The Equations

The equations to be executed at every sample period that will implement the control function of the transfer function in equation 6.1 in order are:

\[
\begin{align*}
 v &= u - w \\
 y &= pv + qw \\
 w &= w + d_i v 
\end{align*}
\]
6.4.4 Software

Now that the equations that need to be implemented are known, the structure of the software for the CSP that will implement these equations using the CSP instruction set can be devised. Since it has been assumed that all the necessary coefficients will be stored on-chip, the first task for the program upon power up would be to move the coefficients to the on-chip coefficient registers.

After this, the processor would await a timing signal/interrupt from the ADC subsystem. This is because it has been assumed for the initial prototype processor model that the timing for the external interrupts will be generated by off-chip circuitry. However, if this was not the case then on-chip timers would have to be set up such that upon time out they would signal the ADC to initiate conversion and when the ADC has finished conversion, it would then send a signal informing the processor that the data is ready; it is this signal that becomes the timing signal/external interrupt that the processor is awaiting in the simple prototype. Upon receiving the external interrupt, the CSP goes off and looks at the address in the IAR, which for the prototype happens to be address location 02H. It is at this address that the ISR starts. So at this address the instructions for the ISR should commence. What has been done, however, is to put a jump instruction at this location and the instructions for the actual ISR are at the jump location.

Simply, the ISR implements the equations of section 6.4.3. Before the calculations can be started, the data from the ADC subsystem must be read ($u$). When this has been done, $v$ is calculated and stored (for use when calculating the new value for $w$), then $y$ is calculated and output, $w$ calculated and stored (for the next sample period) and finally a return from interrupt is executed. A flowchart for the software for the main program and the ISR are given in Figure 6.9.
The code (assembly and machine) for the software of the phase advance example described herewith is given in Appendix J. It is the machine code that is ultimately stored in the EPROM and a close inspection of the EPROM model in Appendix I will confirm that the initial values of the array are the machine code statements.

6.4.5 Comparison with MATLAB

The output from the VHDL simulation was compared with a similar output from a MATLAB simulation of the phase advance controller. Before the comparison of the
results is made, the following points are important to note:

- The VHDL simulation generates a 12-bit output - remember that the input/output wordlength has been chosen to be 12 bit - and therefore to obtain numbers equivalent to the MATLAB output, the VHDL output is divided by 4095 (2^12-1).

- For the MATLAB simulation, the quantised values of the coefficients were used to obtain the results. This would therefore eliminate any differences due to coefficient quantisation.

The sample frequency for the results was 1MHz and the ADC subsystem generates an interrupt signal every 1μs. After the handshaking, the ADC subsystem outputs a 1 onto the data bus (the step input). This is then processed by the processor and the software in the processor controls this. The result from the processor is output to the DAC subsystem. In the DAC subsystem, this output is written to a file for later use. The results from the two simulations compare favourably and there is hardly any noticeable difference between the two as is seen from the response of the controller to a unit step input in Figure 6.10. The differences that exist are minute and are shown in Figure 6.13. Note that for the above example not all the 12 underflow bits that the processor provides are needed. This is the reason there is such a close match between the two simulation outputs.

Also compared were the results of the two simulations in response to a unit negative step input and a step input of high magnitude. These two inputs were chosen to ensure that the VHDL simulation would cope with negative numbers and also to determine how it coped when a large input demand was placed on it.
Figure 6.10 - Comparing responses of a phase advance controller to a positive unit step input under MATLAB and VHDL.

The results for both inputs are shown in Figures 6.11 and 6.12 respectively. As can be seen from the two figures, the results of the two simulations are almost identical and so the VHDL description developed is able to cope with a large range of inputs (unit positive, unit negative and high positive steps).

The plots in Figures 6.10, 6.11 and 6.12 do not show much difference between the responses. Therefore, the difference between the MATLAB and VHDL responses is plotted in Figure 6.13. This shows that the error is quite small (the MATLAB response being the correct one) and the difference is mainly due to the fact that MATLAB uses floating point numbers to calculate the intermediate variables while the VHDL calculations are limited to 24-bits for the internal variables.
A Model for the CSP

Figure 6.11 - Comparing responses of a phase advance controller to a negative unit step input under MATLAB and VHDL.

Figure 6.12 - Comparing responses of a phase advance controller to a high positive unit step input under MATLAB and VHDL.
65 Synthesis

VHDL is such that it allows one to describe the model of a circuit through its behaviour without having to worry about the structure of the model itself; that is questions like "What individual elements will be necessary in the circuit?" and "How will they be connected in order that the required operation can be performed?" can be ignored initially. The behavioural model can in some cases be synthesised such that the structure of the processor is inferred from it. Not all the VHDL constructs can be synthesised and the quality of the synthesis is largely dependent on the synthesis software.

For a VHDL model to be fully synthesisable, the model should be written in structural form. In this way the structure of the circuit can be directly inferred from the model. The resulting schematic is a flattened design. Once the structural model has been developed, it is possible to compile, link and simulate the design and eventually with the aid of the appropriate software target the design to silicon level.
Therefore, the next stage in the development of the CSP is to produce a structural description of the model. The structural description is such that all the functional elements of the processor, such as the multiplier, adder, control unit, the register files etc., are instantiated as components and these are then connected together (routed as on a PCB). Due to lack of time it was not possible to produce a structural description of the whole CSP.

However, to understand the synthesis process in detail, the author did undertake a study of producing a 24-bit Adder/Subtractor Unit (ASU), which is one of the functional units of the CSP, and targeting the design such that it can be implemented on a XILINX FPGA. By targeting the design, the synthesis software is being forced to use parts from the libraries supplied by XILINX for its FPGA's. As was seen in section 5.4.2, at the heart of the ASU is a 24-bit Carry Look Ahead (CLA) Adder. The ASU was modelled in VHDL and the design was verified by simulation. The VHDL description was then synthesised to produce a schematic representation.

It had been hoped to implement and test the ASU on a Xilinx FPGA (Field Programmable Gate Array), but various difficulties were encountered in using the synthesis software which prevented this being achieved within the time available for the project. In retrospect a better method would have been to carry out the entire design process in schematic form and get all the functional units from existing target libraries as provided by silicon vendors, meaning that only the functional units that do not exist or are not adequate for our task would need to be designed from scratch. This method is, on the face of it, far easier, especially if the libraries are well stacked with the appropriate functional units. In fact, for the ASU that was modelled in VHDL, this could have been constructed using the CLA Adder supplied with the XILINX libraries, although the 24-bit width would have to be built using a combination of 16- and 8-bit CLA adders.
6.6 Summary

A behavioural model of the CSP has been developed in VHDL along with a test bench to test the functionality. The basic functionality described in Chapter 5 has been implemented. The 'core' of the processor was designed as specified in Chapter 5. The discussions of Chapter 5 on coefficient format and storage, variable format and storage have been implemented successfully. This was demonstrated by the example on implementing a phase advance controller in section 6.4. It has been shown the responses from the VHDL simulation compare favourably with those obtained from MATLAB simulations.

Due to the limited time available, only one example controller has been demonstrated on the test bench but it should be possible to implement other complex controllers on the architecture designed in VHDL. It is inevitable that further modifications to the design will be required as the one presented in this chapter is only a first cut. Initial attempts at synthesising simple VHDL models have highlighted a few problems that are likely to be encountered when the design progresses to a later stage.
Chapter 7

CONCLUSIONS & SUGGESTIONS

Research into Architectural Considerations for a Control System Processor has been described. The study has concentrated on structures, wordlength requirements of the structures in terms of coefficients and internal variables and requirements of a control system processor.

The δ-operator based modified canonic structure was chosen as a basis for digital controller implementation because of its desirable properties such as low coefficient sensitivity and short internal variable wordlengths. The application of this structure to state space controllers was demonstrated. This application means that common equations result irrespective of the controller representation.

This lead naturally to an investigation of the processor requirements with the view that initially the processor would only be used to implement the controller. It was assumed that any other tasks usually performed in a control system would be performed by off-chip components. A model for a CSP was developed and implemented in VHDL. It was shown that the implementation of a phase advance controller on the model produces results which are comparable to results produced by simulating the controller in MATLAB.

Simulations have been used to confirm the ideas developed within the preceding chapters.
7.1 Contributions of Thesis

The following are the major contributions of the work described in the thesis:

- A comprehensive comparison between two implementation structures for digital controllers in terms of the effects of round-off noise, assessed by three methods.

- The development and assessment of a novel cross-coupled structure based on the δ-operator.

- The combination of the above two leads to an enhanced understanding of practical implementation issues for digital controllers.

- Development of a robust algorithm for producing modified canonic δ-forms for higher order filters, including the evaluation of the coefficient sensitivity in comparison with a z-operator implementation.

- Development of an architecture for the core of a Control System Processor, and its validation using VHDL.

7.2 Conclusions

A number of novel ideas have been explored during the course of this research. The thesis began with a general discussion about the evolution of the microprocessor and digital control. This was followed by an overview presented in Chapter 2 which looked at the field under consideration and identified the areas which required further investigation.
The increasing availability of semi-custom silicon justified further investigation into processors for control. However, the literature survey c. 1991 produced little evidence of research being conducted in the field of architectures for control system processors. The little evidence that the survey produced on processors designed for control needs showed that a vast majority of them were studies in which no hardware was produced. If hardware was produced then it was for processors whose architecture was not entirely generic in nature.

The literature did however produce solid evidence of the advantages of using the $\delta$-operator rather than the widely used $z$-operator. The $\delta$-operator offers superior numerical performance and produces controllers that are highly robust to minor changes in coefficients unlike the $z$-operator based controllers. Additionally, it showed that work on state space controller structures was increasingly centred around the so called 'optimum structures'. The advantages/disadvantages of these structures were discussed.

Chapter 3 looked at the issue of structures in great detail. The reason for this is that, to address the architectural issues of processors, an in-depth study of the underlying digital control algorithms had to be carried out. The algorithms manifest themselves in digital control in the form of structures. For the issue of structures it was decided that the $\delta$-operator based cascaded modified canonic structure would be used. This structure is known to be good and has desirable properties such as the scaling on the internal variables being similar to that of the input - hence eliminating the need for overflow bits. The internal variable wordlength issue was examined in great detail in an attempt to identify a method that would accurately predict the number of underflow bits required for implementation. Two analytical methods were studied: the round-off noise analysis and the steady state analysis.

In round-off noise analysis uniform white noise sources are introduced in the structure whenever there is a truncation/rounding. Since it is statistical in nature, it requires a judgement to be made (e.g. $2\sigma = \text{error tolerable}$). It was shown to
Conclusions and Suggestions

under-predict repeatedly the number of bits required and was therefore deemed overly optimistic. If the results of this method were used blindly, then the resulting controller performance might be seriously degraded and in the extreme there would be a danger of instability in the system.

On the other hand, the steady state analysis assumes worst case condition and hence consistently tends to over-predict the number of underflow bits required. It is pessimistic in nature as its formulation would suggest. It tends to over-predict by 1 or 2 bits at most but this is not a major penalty and the resulting controller would certainly always respond as it was designed to do.

The desire for shorter internal variable wordlengths led to the investigation of combining two first order structures in parallel to produce a second order structure. This was the cross coupled modified canonic structure. A detailed analysis of the structure’s internal variables was carried out. It was shown that once again the overflow of the internal variables would not be an issue.

Additionally, a round-off noise analysis along with a steady state analysis of the structure was performed. The steady state analysis was presented in detail as it had not been done on this structure previously. Once again the results of the earlier section were confirmed - i.e. the round-off noise analysis under-predicts while the steady state analysis over-predicts.

Since the results of the two analyses differed, it was decided that they could not be used when the cascaded modified canonic structure was compared to the cross coupled one. Therefore, the two structures were compared using simulation. A PRBS was used as an input since the sequence could be repeated time after time and also because the resulting output would not have a steady state domination problem when calculating r.m.s errors. The results showed that the savings were not as much as was intuitively expected from the inherent first order nature of the cross coupled structure.
Conclusions and Suggestions

Chapter 4 looked at the application of the modified canonic structure to state space representations. An algorithm which could be used to achieve this was presented. However the algorithm was found to be numerically unstable especially when the eigenvalues of the controller were close to each other. Hence an alternative method was sought and found. This method was found to be highly stable and was demonstrated on an example. The sensitivity of the resulting modified canonic state space controller to changes in coefficients was investigated. To perform the sensitivity analysis, a new semi-analytical method was described. The successful application of the modified canonic structure to state space representation has great implications for processor architectures.

Since a common structure can be used to implement the controller, the resulting equations (and hence software for controllers) are identical. This means that by examining the requirements of the structure, the architecture of the processor could be streamlined to achieve optimum performance. Typical requirements were discussed in Chapter 5 and these included:

- coefficient and variable formats
- instruction set for the processor
- the multiplication routine that will result due the different formats

A novel format for the coefficient was proposed. This format is based on the knowledge gained in implementing controllers in the modified canonic structure. The multiplication routine described is based on asymmetrical words and involves the multiplication of an unsigned coefficient with a signed variable. To the authors knowledge this has not been done in the past.

Chapter 6 developed a model for the CSP based on the principles of Chapter 5. The model was designed in VHDL which allows one to test concepts before building any hardware. A test bench was constructed in VHDL to test the model. Successful implementation of a phase advance controller on the test bench confirmed the ideas of Chapter 5 and the results of the VHDL simulation were compared to those from MATLAB. It was shown that the controller responded properly to a range of inputs: a positive unit step, a negative unit step and a high
positive step. Due to lack of time, no other controllers could be demonstrated but implementation of a range of other controllers would be straightforward. Modifications to the design are inevitable as the design presented is at a very early stage of development. No hardware was produced - the validation was all done in software.

To produce hardware from a VHDL design, one would have to synthesise the VHDL description. An initial investigation into synthesis shows that this process is not straightforward. A simple adder/subtractor unit was designed in VHDL and then a schematic produced by synthesis. Due to various difficulties encountered with the software, no hardware was eventually produced.

7.3 Suggestions for Further Work

The research has achieved many of the objectives set out in Chapter 2. However, the limited time available has meant that many of the ideas could not be explored further. The following sets out briefly some suggestions for further work.

- The assumptions made during the coefficient calculation for the cross coupled structure was just one of the many possibilities. Other possibilities exist and should be explored such that the wordlength requirement can be minimised. To do this, consider the results of the steady state analysis for the cross coupled structure. It showed that the number of bits required for the internal variable is dependent on the smaller of the set \( [d_1, d_2] \). Therefore, one could use an optimisation approach to determine the coefficients of the cross coupled structure which would minimise the wordlength requirements.
Conclusions and Suggestions

- In Chapter 4, a modified canonic state space structure was presented. This extension of the modified canonic structure was made for single input systems (i.e. SISO and SIMO). It has been tested successfully for these systems but no work was done for multiple input systems (MISO, MIMO). This is an area where further work is required.

- The internal variable wordlength requirements for the high order state space controller were not explored and should be to confirm that the results of Chapter 3 and other work done by [Forsythe and Goodall (1991)] still applies.

- In Chapter 5 an instruction set was presented for a CSP. Various enhancements to this would be useful, for example the provision of left shifting to enable gains greater than unity to be implemented. It is inevitable that other instructions may also be required to cope with other as yet unexplored functions. Therefore, it is recommended that the instruction set be reviewed and extensions to it made as necessary.

- The CSP model does not include any provision for overflow handling. If gains greater than 1 are going to be accommodated with the addition of left shifting, as recommended above, then overflow handling should also be designed into the system.

- The model for the CSP presented in Chapter 6 was an initial design and therefore does not manage interrupts in the traditional manner. It assumes that other off-chip circuitry is available to cope with timer/counter type tasks. This timer/counter capability should be added to the model. The model could be further enhanced such that interrupt handling is more representative of what happens on the chip in practice.
Conclusions and Suggestions

- Investigate the time taken (in real terms) to process the algorithm. That is, how many milliseconds or microseconds of cpu time does it require to execute different orders of algorithms with the current design and hence what is the maximum achievable sample frequency for these orders. Based on these results further modifications to the architecture, communication protocols, clock phases, bus polling etc. can be made.


References


Jacklin, S. A. et. al. (1986). 'Integrating computer architectures into the design of high-performance controllers', *IEEE Control Systems Mag.*, No. 6, pp 3-8.


References


Appendix A

TRANSFER FUNCTIONS

The transfer functions for second order lowpass and notch filters are given below. These transfer functions are in general form. For implementation purposes, the emulation tables given in [Forsythe and Goodall (1991)] are used to obtain the equivalent transfer functions in δ.

**Notch Filter**

\[
H_N(s) = \frac{1 + \frac{s^2}{\omega_n^2}}{1 + \frac{2\zeta s}{\omega_n} + \frac{s^2}{\omega_n^2}}
\]

**Lowpass Filter**

\[
H_L(s) = \frac{1}{1 + \frac{2\zeta s}{\omega_o} + \frac{s^2}{\omega_o^2}}
\]

where \(\omega_o\) is the corner frequency of the filter in rad s\(^{-1}\) and \(\zeta\) is the damping factor of the filter.
Appendix B

**RMS TRANSFER FUNCTION MAGNITUDES**

As was seen in Section 3.1.2.1, calculation of the r.m.s transfer function magnitude involves solving a contour integral of the type

\[
\frac{1}{2\pi j} \oint H(z)H(z')z' dz
\]  

(B.1)

The above integral can be evaluated in a number of different ways as outlined in [Mitra et. al. (1974)]. The four methods considered here are

- Cauchy's residue theorem
- Mitra, Hirano and Sakaguchi's simple method as outlined in [Mitra et. al. (1974)]
- rewriting the contour integral as

\[
\frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} |H(e^{j\omega}f)|^2 d\omega
\]  

(B.2)

and using numerical integration

- As an alternative, H(z) could be emulated to H(s) using the bilinear transform (any other emulation technique could also be used but the bilinear transform was favoured because it is simple and effective) and then obtaining a closed form solution of

\[
\frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} |H(j\omega)f|^2 d\omega
\]  

(B.3)

The solutions of above methods are presented in the section B.1, for the modified canonic form.
The modified canonic form with noise sources is as shown in Figure 3.2. The first step in calculating the r.m.s transfer function magnitudes is to find the transfer functions between the various noise sources, $e_1$, $e_2$, and $e_3$, and the output, $y$. The transfer functions can easily be calculated using Mason's rule [Kuo (1987)].

The transfer function $H_1(z)$ between the noise source $e_1(n)$ and $y$, can be shown to be

$$H_1(z) = \frac{z(p_3 - p_1) + [(p_2 - p_1)d_2 - (p_2 - p_1)]}{z^2 - z(2 - d_1 + d_1d_2) + (1 - d_1 + d_1d_2)}$$

(B.4)

Similarly, $H_2(z)$ between $e_2(n)$ and $y$ can be shown to be

$$H_2(z) = \frac{z(p_1 - p_2) + [(p_2 - p_1)d_1 - (p_2 - p_1)]}{z^2 - z(2 - d_1 + d_1d_2) + (1 - d_1 + d_1d_2)}$$

(B.5)

and $H_3(z)$ between $e_3(n)$ and $y$ is

$$H_3(z) = 1$$

(B.6)

It can be shown that $H_1(z)$ and $H_2(z)$ have poles

$$z_1 = re^{j\theta} \quad \text{and} \quad z_2 = re^{j\theta}$$

(B.7)

where

$$r = \sqrt{1 - d_1 + d_1d_2} \quad \text{and} \quad \tan(\theta) = \frac{4d_1d_2 - d_1^2}{(2 - d_1)^2}$$

(B.8)

Hence, the transfer functions in (B.4) and (B.5) can be written as

$$H_1(z) = \frac{z(p_3 - p_1) + [(p_2 - p_1)d_2 - (p_2 - p_1)]}{(z - re^{j\theta})(z - re^{j\theta})}$$

(B.9)

$$H_2(z) = \frac{z(p_1 - p_2) + [(p_2 - p_1)d_1 - (p_2 - p_1)]}{(z - re^{j\theta})(z - re^{j\theta})}$$

(B.10)
B.II Cauchy’s Residue Theorem

The solution of $H_{ir}^2$, $H_{2r}^2$, and $H_{3r}^2$ using Cauchy’s Residue theorem is arrived at in this section. The method can be algebraically tedious and, hence, highly error prone. It is outlined briefly using $H_{ir}^2$ as an example.

$H_{ir}^2$
The r.m.s transfer function magnitude $H_{ir}^2$ can be evaluated by solving the following equation

$$H_{ir}^2 = \frac{1}{2\pi j} \int f(z) \, dz \tag{B.11}$$

where $f(z) = H_1(z) H_1(z^{-1}) z^i$

$H_1(z)$ is given by equation (B.9) and $H_1(z^{-1})$ can be obtained by substituting $z$ with $z^i$ in the same equation and after manipulation, $f(z)$ can be shown to be

$$f(z) = \frac{z^i( p_2 - p_1)B + z^i( p_2 - p_1) \gamma + B^2 + B( p_2 - p_1) }{(z - re^{j\theta})(z - re^{j\theta})(1 - ze^{j\theta})(1 - ze^{j\theta})} \tag{B.12}$$

where $B = (p_3 - p_i)d_z - (p_2 - p)$

By Cauchy’s residue theorem, the solution of the contour integral in (B.11) is given by:

$$\text{Integral} = 2\pi j \sum (\text{Residues of all poles inside unit circle}) \tag{B.13}$$

therefore, only the poles of $f(z)$ that are inside the unit circle need to be considered.

Assuming $H_1(z)$ is stable, these poles are when $z = re^{j\theta}$ and $z = re^{j\theta}$. Hence, need to find the residues of $f(z)$ when $z = re^{j\theta}$ and $z = re^{j\theta}$. The residue of $f(z)$ at $z = a$ can be evaluated using the formula

$$\text{Residue} = \lim_{z \to a} (z - a) f(z) \tag{B.14}$$

Using (B.14) and after simplification, it can be shown that Residue, when $z = re^{j\theta}$ is given by

Architectural Considerations for a CSP
RMS Transfer Function Magnitudes

Residue_1 = \frac{x_{n1} + jy_{n1}}{2r(1 - r^2)(x_{d1} + jy_{d1})}

and Residue_2 when z = re^{j\theta} is given by

Residue_2 = \frac{x_{n1} - jy_{n1}}{2r(1 - r^2)(x_{d1} - jy_{d1})}

and using (B.13), the integral, the sum of Residue_1 and Residue_2 is

\text{Integral} = \frac{2\pi j}{r(1-r^2)} \left[ \frac{x_{n1}x_{d1} + y_{n1}y_{d1}}{x_{d1}^2 + y_{d1}^2} \right]

And finally \( H^2_{1R} \) can be evaluated to be

\begin{equation}
H^2_{1R} = \frac{1}{r(1-r^2)} \left[ \frac{x_{n1}x_{d1} + y_{n1}y_{d1}}{x_{d1}^2 + y_{d1}^2} \right]
\end{equation}

where

\begin{align*}
x_{n1} &= B_1 r \cos(\theta) + B_2 r' \cos(2\theta) + B_2 \\
y_{n1} &= B_1 r \sin(\theta) + B_2 r' \sin(2\theta) \\
x_{d1} &= r' \sin(\theta) \sin(2\theta) \\
y_{d1} &= \sin(\theta) - r' \sin(\theta) \cos(2\theta)
\end{align*}

and

\begin{align*}
B_1 &= (p_3 - p_1)^2 + [(p_3 - p_2) + d_2 - (p_2 - p_1)]^2 \\
B_2 &= (p_3 - p_1)(p_3 - p_2) + d_2 - (p_2 - p_1)
\end{align*}

with \( r \) and \( \theta \) are as given in equation (B.8).

\( H^2_{2R} \)

Similarly it can be shown that \( H^2_{2R} \) is given by

\begin{equation}
H^2_{2R} = \frac{1}{r(1-r^2)} \left[ \frac{x_{n2}x_{d2} + y_{n2}y_{d2}}{x_{d2}^2 + y_{d2}^2} \right]
\end{equation}

where

\begin{align*}
x_{n2} &= C_1 r \cos(\theta) + C_1 r' \cos(2\theta) + C_1 \\
y_{n2} &= C_1 r \sin(\theta) + C_1 r' \sin(2\theta) \\
x_{d2} &= r' \sin(\theta) \sin(2\theta) \\
y_{d2} &= \sin(\theta) - r' \sin(\theta) \cos(2\theta)
\end{align*}
and
\[ C_1 = (p_3 - p_1)^2 + [(p_3 - p_2)d_1 - (p_3 - p_1)]^2 \]
\[ C_2 = (p_3 - p_1)[(p_3 - p_2)d_1 - (p_3 - p_1)] \]

Once again, \( r \) and \( \theta \) are as given in equation (B.8).

\( H_{3r}^2 \)

The solution for \( H_{3r}^2 \) is trivial and can be easily shown to be
\[ H_{3r}^2 = 1 \]

B.1.2 Mitra et al.'s Simple Method

This method is indeed very simple and easy to use. It can be error prone if the procedure is not understood properly. Unfortunately, the author only became aware of the method in the later stages of the research.

The method makes use of the partial fraction expansion of a transfer function \( H(z) \) and the result of the contour integral in (B.1) can easily be obtained by referring to equation (7) and Table 1 in [Mitra et al. (1974) pp 326 and pp 328 respectively].

The solution for \( H_{3r}^1 \) and \( H_{3r}^2 \) is easily arrived at (in \( 1/2 \) a side compared to 10 sides or so it took for Cauchy's residue method) and is as given in the equations below.

\[ H_{3r}^2 = \frac{(b_1^2 + b_0^2)(1 - a_0^2) - 2a_1b_1b_0(1 - a_0)}{(1 - a_0^2)^2 + 2a_0a_1^3 - a_1^2(1 + a_0^2)} \quad (B.17) \]

where
\[ b_1 = p_2 - p_1 \]
\[ b_0 = (p_3 - p_2)d_2 - (p_3 - p_1) \]
\[ a_1 = -(2 - d_1) \]
\[ a_0 = 1 - d_1 + d_1d_2 \]

\[ H_{3r}^2 = \frac{(c_1^2 + c_0^2)(1 - a_0^2) - 2a_1c_1c_0(1 - a_0)}{(1 - a_0^2)^2 + 2a_0a_1^3 - a_1^2(1 + a_0^2)} \quad (B.18) \]

where
\[ c_1 = p_3 - p_1 \]
\[ c_0 = (p_3 - p_2)d_1 - (p_3 - p_1) \]

Architectural Considerations for a CSP
and \( a_i \) and \( a_o \) are as before.

\( H_{2R} \) is as before.

**B.I.3 Numerical Integration**

Numerical integration of equation (B.2) for \( H_{1R}^2 \) and \( H_{2R}^2 \) can easily be performed using MATLAB. \( |H(e^{j\omega t})| \) is the magnitude of \( H(z) \) and can be obtained by using the `DBODE` function while integration can be performed using the `TRAPZ` function in MATLAB.

**B.I.4 Emulation to H(s)**

This alternative was investigated with the hope that the solution of equation (B.3) might be easier, less tedious algebraically and, therefore, less error prone when compared with the solution of equation (B.1) that is obtained using Cauchy's residue theorem.

The first step is to emulate \( H(z) \) to \( H(s) \). The bilinear transform was used to perform the emulation and after simplification it can be shown that

\[
H_1(s) = \frac{n_{01} + n_{11}s + n_{21}s^2}{m_0 + m_1s + m_2s^2} \quad (B.19)
\]

\[
H_2(s) = \frac{n_{02} + n_{12}s + n_{22}s^2}{m_0 + m_1s + m_2s^2} \quad (B.20)
\]

where

\[
\begin{align*}
n_{01} &= (p_3 - p_1)d_2 \\
n_{11} &= T[(p_2 - p_1) - (p_3 - p_1)d_2] \\
n_{21} &= \frac{V}{4} T^2[(p_2 - p_1) + 2(p_2 - p_1)] \\
m_0 &= d_d \\
m_1 &= Td_1[1 - d_d] \\
m_2 &= \frac{V}{4} T^2[4 - 2d_1 + d_d] \\
n_{02} &= (p_3 - p_2)d_2 \\
n_{12} &= T[(p_3 - p_2) - (p_3 - p_1)d_2] \\
n_{22} &= \frac{V}{4} T^2[(p_3 - p_2) + 2(p_3 - p_1)] \\
m_0 &= d_d \\
m_1 &= Td_1[1 - d_d] \\
m_2 &= \frac{V}{4} T^2[4 - 2d_1 + d_d]
\end{align*}
\]
The r.m.s transfer function magnitude can be evaluated by solving equation (B.3). The method is illustrated using $H^2_{1r}$ as an example.

$H^2_{1r}$

First, the frequency response of $H_1(s)$ is obtained by substituting $s=j\omega$ in (B.19) and the square of the magnitude of $H_1(j\omega)$ and is given by

$$|H_1(j\omega)|^2 = \frac{n_{01}^2 + (n_{21}^2 - 2n_{01}n_{21})\omega^2 + n_{21}^2\omega^4}{m_0^2 + (m_1^2 - 2m_0m_2)\omega^2 + m_2^2\omega^4}$$

and the r.m.s transfer function magnitude is given by

$$H^2_{1r} = \frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} \frac{n_{01}^2 + (n_{21}^2 - 2n_{01}n_{21})\omega^2 + n_{21}^2\omega^4}{m_0^2 + (m_1^2 - 2m_0m_2)\omega^2 + m_2^2\omega^4} d\omega$$

The solution of the integration is not as simple as it may seem. The first step is to reduce the order of the numerator by diving out the denominator. After further rearranging:

$$H^2_{1r} = I_{11} + I_{21} + I_{31}$$

where

$$I_{11} = \frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} \frac{n_{21}^2}{m_2^2} d\omega$$

$$I_{21} = \frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} \frac{N_{11}\omega^2}{m_0^2 + m_{11}\omega^2 + m_2^2\omega^4} d\omega$$

$$I_{31} = \frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} \frac{N_{01}}{m_0^2 + m_{11}\omega^2 + m_2^2\omega^4} d\omega$$

and

$$m_{11} = m_1^2 - 2m_0m_2$$

$$N_{11} = n_{21}^2 - 2n_{01}n_{21} - \frac{n_{21}^2m_{11}}{m_2^2}$$

$$N_{01} = n_{01}^2 - \frac{n_{01}n_{21}^2}{m_2^2}$$

The solution of $I_{11}$ is straightforward and is

$$I_{11} = \frac{n_{21}^2}{m_0^2}$$
The solutions of $I_n$ and $I_m$ are not as straightforward and the symbolic processor on Mathcad (v 4.0) was used to obtain the appropriate answers which after simplification are as given below

$$I_{21} = -\frac{T N_{11}}{4\pi m_2 m_3} \ln \left( \frac{T \pi m_3 + T^2 m_0 + \pi^2 m_2}{-T \pi m_3 + T^2 m_0 + \pi^2 m_2} \right)$$

$$+ \frac{T N_{11}}{2\pi m_2 m_4} \left( \tan \left( \frac{2\pi m_2 + T m_3}{T m_4} \right) - \tan \left( \frac{-2\pi m_2 + T m_3}{T m_4} \right) \right)$$

$$I_{31} = \frac{T N_{11}}{4\pi m_0 m_3} \ln \left( \frac{T \pi m_3 + T^2 m_0 + \pi^2 m_2}{-T \pi m_3 + T^2 m_0 + \pi^2 m_2} \right)$$

$$+ \frac{T N_{11}}{2\pi m_0 m_4} \left( \tan \left( \frac{2\pi m_2 + T m_3}{T m_4} \right) - \tan \left( \frac{-2\pi m_2 + T m_3}{T m_4} \right) \right)$$

where $m_3 = \sqrt{2m_2 m_0 - m_{11}}$ and $m_4 = \sqrt{2m_2 m_0 + m_{11}}$

$H_{2r}$

Following a similar procedure for $H_{2r}$, the following can be shown easily.

$$H_{2r}^2 = I_{12} + I_{22} + I_{32}$$

where $I_{12} = \frac{n_{22}^2}{m_2^2}$

$$I_{22} = -\frac{T N_{12}}{4\pi m_2 m_3} \ln \left( \frac{T \pi m_3 + T^2 m_0 + \pi^2 m_2}{-T \pi m_3 + T^2 m_0 + \pi^2 m_2} \right)$$

$$+ \frac{T N_{12}}{2\pi m_2 m_4} \left( \tan \left( \frac{2\pi m_2 + T m_3}{T m_4} \right) - \tan \left( \frac{-2\pi m_2 + T m_3}{T m_4} \right) \right)$$

$$I_{32} = \frac{T N_{12}}{4\pi m_0 m_3} \ln \left( \frac{T \pi m_3 + T^2 m_0 + \pi^2 m_2}{-T \pi m_3 + T^2 m_0 + \pi^2 m_2} \right)$$

$$+ \frac{T N_{12}}{2\pi m_0 m_4} \left( \tan \left( \frac{2\pi m_2 + T m_3}{T m_4} \right) - \tan \left( \frac{-2\pi m_2 + T m_3}{T m_4} \right) \right)$$
and \[ N_{12} = n_{12}^2 - 2n_{01}n_{22} - \frac{n_{22}^2 m_{11}}{m_2^2} \]

\[ N_{02} = n_{02}^2 - \frac{m_0^2 n_{22}^2}{m_2^2} \]

with \( m_{11}, m_1, \) and \( m_2 \) as for \( H_{12}^2 \).

**B.15 Comparison of the 4 methods**

The accuracy of the analytical solutions of sections B.1.1, B.1.2 and B.1.4 can be checked by comparing the results against the results obtained by performing numerical integration, on equation (B.2) for the case of sections B.1.1 and B.1.2, and, on equation (B.3) for the case of section B.1.4. The results are given in Table B.1 for a 10 Hz lowpass filter, damping factor \( \zeta = 0.707 \), sampled at 100 Hz. Note NI stands for numerical integration in the table.

<table>
<thead>
<tr>
<th></th>
<th>Cauchy's</th>
<th>Mitra et al</th>
<th>NI - (B.2)</th>
<th>Emulation</th>
<th>NI (B.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H_{12}^2 )</td>
<td>0.29474</td>
<td>0.29474</td>
<td>0.29474</td>
<td>0.31905</td>
<td>0.31905</td>
</tr>
<tr>
<td>( H_{12}^1 )</td>
<td>1.688</td>
<td>1.688</td>
<td>1.6879</td>
<td>1.8697</td>
<td>1.8696</td>
</tr>
</tbody>
</table>

*Table B.1 Comparison of Analytical solution and Numerical Integration*

From the table it is clear that the results of Cauchy’s residue theorem and those of Mitra et al. are correct as they match the numerical integration results (columns 2, 3 and 4) while columns 5 and 6 confirm the validity of the analytical results from the emulation method.
The transfer function of the cross coupled structure as given in equation (3.14) is derived in this appendix and the reader might need to refer to Figure 3.13 to see how some of the equations arise. The equations for the structure are:

\[ v_1 = u - w_1 \]
\[ v_2 = u - w_2 \]

and in vector form this can be rewritten as

\[
\begin{bmatrix}
  v_1 \\
  v_2
\end{bmatrix} =
\begin{bmatrix}
  I \\
  I
\end{bmatrix} u -
\begin{bmatrix}
  w_1 \\
  w_2
\end{bmatrix}
\]

(C.1)

\[ v = u - w \]

Similarly

\[ w_1 = w_1 + (d_{11} v_1 + d_{12} v_2) \]
\[ w_2 = w_2 + (d_{21} v_1 + d_{22} v_2) \]

can be written in vector form as

\[
\begin{bmatrix}
  w_1 \\
  w_2
\end{bmatrix} =
\begin{bmatrix}
  w_1 \\
  w_2
\end{bmatrix} +
\begin{bmatrix}
  d_{11} & d_{12} \\
  d_{21} & d_{22}
\end{bmatrix}
\begin{bmatrix}
  v_1 \\
  v_2
\end{bmatrix}
\]

(C.2)

\[ w = w + d_{11} v = (d_{11} v) \delta^{-1} \]

and

\[ y = p_1 v_1 + p_2 v_2 + q_1 w_1 + q_2 w_2 \]

can be written as
Transfer function of Cross Coupled Structure

\[ y = \left[ p_1 \left[ \begin{array}{c} v_1 \\ v_2 \end{array} \right] + \left[ q_1 \left[ \begin{array}{c} w_1 \\ w_2 \end{array} \right] \right] \right. \]
\[ = p \left[ \begin{array}{c} v_1 \\ v_2 \end{array} \right] + q \left[ \begin{array}{c} w_1 \\ w_2 \end{array} \right] \]  \hspace{1cm} (C.3)

Substituting (C.2) into (C.1) gives

\[ v = u - (d_{ij} \delta)^{-1} \]
\[ v + (d_{ij} \delta)^{-1} = u \]
\[ (I + d_{ij} \delta)^{-1} v = u \]
\[ v = (I + d_{ij} \delta)^{-1} u \]  \hspace{1cm} (C.4)

Similarly by substituting (C.2) into (C.3), it can be shown

\[ y = (p + q d_{ij} \delta)^{-1} v \]  \hspace{1cm} (C.5)

So substituting (C.4) into (C.5) gives

\[ y = (p + q d_{ij} \delta)^{-1} (I + d_{ij} \delta)^{-1} u \]  \hspace{1cm} (C.6)

Expanding the first term on the right hand side of (C.5) gives

\[ (p + q d_{ij} \delta)^{-1} = \left[ (p_i + q_1 d_{ij} \delta + q_2 d_{ij} \delta^{-1}) \quad (p_2 + q_1 d_{ij} \delta + q_2 d_{ij} \delta^{-1}) \right] \]  \hspace{1cm} (C.7)

and the inverse term of (C.5) can be shown to be

\[ (I + d_{ij} \delta)^{-1} = \frac{1}{\Delta} \left[ \begin{array}{cc} 1 + d_{ij} \delta^{-1} & -d_{ij} \delta^{-1} \\ -d_{ij} \delta^{-1} & 1 + d_{ij} \delta^{-1} \end{array} \right] \]  \hspace{1cm} (C.8)

where \( \Delta \) is the determinant and can be shown to be

\[ \Delta = 1 + (d_{ij} + d_{ij}) \delta^{-1} + (d_{ij} d_{ij} + d_{ij} d_{ij}) \delta^{-2} \]  \hspace{1cm} (C.9)

and finally remember that

\[ u = \left[ \begin{array}{c} 1 \\ 1 \end{array} \right] u \]  \hspace{1cm} (C.10)

Finally to derive equation (3.14) equations (C.7), (C.8), (C.9) and (C.10) need to be substituted into (C.6) and the terms rearranged to get the desired transfer function.
To perform the initial and final value analysis on the internal variables of the cross coupled system, we first need to find the transfer functions that relate the internal variables to the input \( u \). For the analysis presented below, the reader might need to refer to Figure 3.13 which depicts the cross coupled structure.

The equations used to calculate the internal variables are:

\[
\begin{align*}
  v_1 &= u - w_1 \\
  v_2 &= u - w_2 \\
  w_1 &= w_1 + (d_{11}v_1 + d_{12}v_2) \\
  &= (d_{11}v_1 + d_{12}v_2)\delta^{-t} \\
  w_2 &= w_2 + (d_{21}v_1 + d_{22}v_2) \\
  &= (d_{21}v_1 + d_{22}v_2)\delta^{-t}
\end{align*}
\]

Substituting equation (D.3) into (D.1) and rearranging, gives

\[
v_1 + d_{11}v_1\delta^{-t} + d_{12}v_2\delta^{-t} = u \tag{D.5}
\]

and substituting (D.4) into (D.2) and making \( v_2 \) the subject gives

\[
v_2 = \frac{u - d_{21}v_1\delta^{-t}}{1 + d_{22}\delta^{-t}} \tag{D.6}
\]

by substituting (D.6) into (D.5) and rearranging we can show that

\[
\frac{v_1}{u} = \frac{l + (d_{22} - d_{12})\delta^{-t}}{l + (d_{11} + d_{22})\delta^{-t} + (d_{11}d_{22} - d_{12}d_{21})\delta^{-2}} \tag{D.7}
\]

By making \( v_1 \) the subject of (D.5) and substituting the result into (D.6) we can show that
\[
\frac{v_2}{u} = \frac{1 + (d_{11} - d_{12})\delta^{-1}}{1 + (d_{11} + d_{22})\delta^{-1} + (d_{11}d_{22} - d_{12}d_{21})\delta^{-2}}
\]  
(D.8)

For \( w_1/u \), we need to substitute (D.7) and (D.8) into (D.3). After further manipulation, we can show that

\[
\frac{w_1}{u} = \frac{(d_{11} + d_{12})\delta^{-1} + (d_{11}d_{22} - d_{12}d_{21})\delta^{-2}}{1 + (d_{11} + d_{22})\delta^{-1} + (d_{11}d_{22} - d_{12}d_{21})\delta^{-2}}
\]  
(D.9)

and similarly we can show that

\[
\frac{w_2}{u} = \frac{(d_{21} + d_{22})\delta^{-1} + (d_{11}d_{22} - d_{12}d_{21})\delta^{-2}}{1 + (d_{11} + d_{22})\delta^{-1} + (d_{11}d_{22} - d_{12}d_{21})\delta^{-2}}
\]  
(D.10)

Having obtained the relationships between the internal variables and the input the initial and final values of the internal variables in response to an input \( u = u_{\text{max}} \) can now be determined. The initial value is when \( \delta^i = 0 \) and the final value is when \( \delta^f \rightarrow \infty \).

\( v_1 \) using equation (D.7)

\[
\delta^i = 0 \quad : \quad \frac{v_1}{u} = \frac{1}{1} = 1
\]

\[
\delta^i \rightarrow \infty \quad : \quad \frac{v_1}{u} \rightarrow \frac{1}{(d_{11}d_{22} - d_{12}d_{21})\delta^{-2}} \rightarrow 0
\]

\( v_2 \) using equation (D.8)

\[
\delta^i = 0 \quad : \quad \frac{v_2}{u} = 1
\]

\[
\delta^i \rightarrow \infty \quad : \quad \frac{v_2}{u} \rightarrow \frac{1}{(d_{11}d_{22} - d_{12}d_{21})\delta^{-2}} \rightarrow 0
\]

\( w_1 \) using equation (D.9)

\[
\delta^i = 0 \quad : \quad \frac{w_1}{u} = \frac{0}{1} = 0
\]

\[
\delta^i \rightarrow \infty \quad : \quad \frac{w_1}{u} \rightarrow \frac{(d_{11}d_{22} - d_{12}d_{21})\delta^{-2}}{(d_{11}d_{22} - d_{12}d_{21})\delta^{-2}} \rightarrow 1
\]
Initial and Final Value Analysis

\[ w_i \text{ using equation (D.10)} \]

\[ \delta^i = 0 : \quad \frac{w_2}{u} = \frac{0}{l} = 0 \]

\[ \delta^i \rightarrow \infty : \quad \frac{w_2}{u} \rightarrow \frac{\left(d_{11}d_{22} - d_{12}d_{21}\right)\delta^{-2}}{\left(d_{11}d_{22} - d_{12}d_{21}\right)\delta^{-2}} \rightarrow l \]
The relationship between $d_n$ and $a_n$ is as given in equation (4.11). This relationship was arrived at by equating equations (4.9) and (4.10) for different orders of systems, eg 2nd, 3rd, 4th, 5th, etc., and examining the trends through the different orders. The resulting algebraic expression for the different orders has a common format and it can be grouped into the form

$$a_n = T_d d_n + c_n$$

This appendix presents the $T_d$ matrix and the $c_n$ vector for the different orders.

From the different orders we should be able to determine a trend and therefore formulate a $T_d$ matrix and a $c_n$ vector for any order of system.

For a second order structure the following relationship can be shown:

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_1 d_2 \end{bmatrix} + \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Similarly for a third order system we can show

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_1 d_2 \\ d_1 d_2 d_3 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix}$$

and for a fourth order system

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_1 d_2 \\ d_1 d_2 d_3 \\ d_1 d_2 d_3 d_4 \end{bmatrix} + \begin{bmatrix} 4 \\ 6 \\ 4 \\ 1 \end{bmatrix}$$
and a fifth order system produces

\[
\begin{bmatrix}
  a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5
\end{bmatrix}
= \begin{bmatrix}
  1 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \\ 6 & 3 & 1 & 0 & 0 \\ 4 & 3 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
  d_1 \\ d_1d_2 \\ d_1d_2d_3 \\ d_1d_2d_3d_4 \\ d_1d_2d_3d_4d_5
\end{bmatrix}
+ \begin{bmatrix}
  5 \\ 10 \\ 10 \\ 5 \\ 1
\end{bmatrix}
\]

and so on for higher order systems.

It should be noted that the elements of the $T_d$ matrix are the entries of Pascal's triangle and the entries of the $c_a$ vector are also from Pascal's triangle. Pascal's triangle is

\begin{align*}
1 \\
1 & 1 \\
1 & 2 & 1 \\
1 & 3 & 3 & 1 \\
1 & 4 & 6 & 4 & 1 \\
1 & 5 & 10 & 10 & 5 & 1 \\
1 & 6 & 15 & 20 & 15 & 6 & 1 \\
1 & 7 & 21 & 35 & 35 & 21 & 7 & 1
\end{align*}

and so on.

So for the fifth order system above, the 1st column of the $T_d$ matrix is the 5th row of Pascal's triangle (if we ignore the signs for the moment) and the 2nd column of $T_d$ is the fourth row of the triangle and so on. For a fifth order system the entries of the $c_a$ vector are from the 6th row of the triangle. We can therefore construct a $T_d$ matrix and a $c_a$ vector for any order of system which needs to be converted from the controller canonical form to the modified canonic form.
function [ad,bd,cd,dd,d,p,That] = z2delta(az,bz,cz,dz);
%   [ad,bd,cd,dd,d,p,That] = z2delta(az,bz,cz,dz);
%
% The matrices az,bz,cz,dz are in the discrete form and
% in controller canonical form. The resulting matrices
% ad,bd,cd,dd are also in discrete form and the coefficients
% conform to the delta form. The vectors d and p give the
% coefficients of the modified canonic structure.
%
% az, bz are assumed to be in the following form
%
% az = [-a1 -a2 -a3 ... -an-1 -an
%       1    0    0    ...    0    0
%       0    1    0    ...    0    0
%       :    :    :    ...    :    :
%       0    0    0    ...    1    0]
% bz = [1 0 0 ... 0]
%
% Now handles Single Input Multi Output Case
% Written by Dipesh.
% Last Modified 11th July 1994.

[row,col] = size(az);
[rowc,colc] = size(cz);
msg = abcdchk(az,bz,cz,dz);
if (msg ~= []),
    error('Matrices not dimensionally consistent');
end;
if (row ~= col),
    error('The "a" matrix should be square');
end;
% First form Td which can be used to relate the a1 ... an to d1 ... dn.
t = pascal(col,1);
Td = zeros(col);

for i = 1:col,
    n = col+1-i;
    Td(:,i) = t(n,1:col)';
end

Td = flipud(Td);
for i = 1:col,
    Td(i:col,i) = flipud(Td(i:col,i));
end

TdInv = inv(Td);

% Now form the Cn vector (called Const here)
temp = pascal(col+1,1);
Const = temp(col+1,2:col+1)';

% Determine the an vector of the form:
% an = [ a1
% a2
% a3
% ...
% an ];
an = -az(1,:);'

% Now calculate the dn vector which is of the form:
% dn = [ d1
d1d2
d1d2d3
% ...
% d1...dn];
dn = TdInv * (an - Const);

d is a vector which gives the coefficients d1 ... dn
% d = [ d1
d2
d3
% ...
% dn ];
Matlab function of Chapter 4 Algorithm

d(1) = dn(1);

for i = 2:col,
    d(i) = dn(i) / dn(i-1);
end;

d = d';

% Now form the adelta matrix; it is of the form
% adelta = [-d1 -d1 -d1 -d1 -d1
%            d2 0 0 ... 0 0
%            0 d3 0 ... 0 0
%            : : : ... : :
%            0 0 0 ... dn 0]..
adelta = diag(d(2:col),-1);
adelta(1,:) = -d(1) * ones(1,col);

% ad = I + adelta
ad = eye(col) + adelta;

% bd = [d1 0 0 0 ... 0]'
bd = zeros(col,1);
bd(1) = d(1);

% Now construct the similarity transformation matrix which
% will transform the modified canonic system to the controllable
% canonic system - see Ogata (Discrete Time Control Systems).

% First Step is to find M (the controllability matrix)
M = ctrb(ad,bd);

% Next step is to form W
temp = poly(ad);
temp = flipr(temp);
W = zeros(col);
for i = 1:col,
    W(i,:) = [temp(1,i+1:col+1) zeros(1,i-1)];
end;

% And finally T
T = M * W;

% The system I have used is related to Ogata's one by the
% transformation R ... so do that now.
R = eye(col);
R = flipr(R);

Architectural Considerations for a CSP
Matlab function of Chapter 4 Algorithm

% Can now find T hat which will transform a controllable
canonic system to a delta - modified canonic system.
That = inv(T * R);

% And finally find the c and d matrices
cd = cz * (That);
dd = dz;

% p is a vector which gives the coefficients p1 ... pn
for i = 1:rowc,
    p(i,1) = dd(i);
    p(i,2:colc+1) = cd(i,:) - (p(i,1) * ones(1,colc));
end;
Appendix G

$H_\infty$ CONTROLLER

The controller used as an example in section 4.7.4 was designed as a continuous controller and the A, B, C, D matrices are:

$$A = \begin{bmatrix}
-3.6225 & -0.2177 & 2.4020 & 0.2741 & -0.0376 & 0 & 0.0238 \\
-0.6630 & -7.6539 & 1.4514 & -1.2488 & -0.9790 & -0.1740 & 0.0267 \\
-2.1555 & 2.2575 & -1.9995 & -0.1831 & 0.3074 & 0.0430 & -0.0447 \\
0.5250 & 0.2179 & 0.7738 & -2.0703 & -2.6843 & -0.4012 & -0.0489 \\
-0.3229 & -0.4059 & -0.3523 & 3.0491 & -4.2599 & -1.5012 & 0.2540 \\
-0.0213 & 0.1532 & -0.0924 & 0.1819 & -0.0189 & -2.5247 & 1.4101 \\
-0.0044 & 0.0689 & -0.0335 & 0.0394 & 0.0710 & -2.5024 & -1.3997 \\
\end{bmatrix}$$

$$B = \begin{bmatrix}
-7.1899 & -3.1372 & -1.0985 & 0.5174 & -0.3666 & 0.0050 & 0.0071 \\
-7.6849 & 1.0424 & 1.9163 & 0.4196 & 0.0521 & 0.0165 & 0.0231 \\
\end{bmatrix}$$

$$C = \begin{bmatrix}
-13.0891 \\
\end{bmatrix}$$

Note that although the original controller was a 2 input 2 output controller, only the single input single output case is given here. When this controller is converted to the discrete one at a sampling frequency of 1 kHz, (using the c2dm function in MATLAB using Tustin's method) the z-operator controller matrices are as given below
The continuous controller is converted to the $\delta$-operator form using the method outlined by [Middleton and Goodwin (1990)]. This is then followed by multiplying the $A_\delta$ and $B_\delta$ matrices by $T$. The resulting system is the converted to transfer function form using the MATLAB function $ss2tf$ and this reconverted to state space form using the MATLAB function $tf2ss$. The resulting system is in controller canonical form:

$$A_\delta = \begin{bmatrix}
-2.35e-2 & -2.36e-4 & -1.38e-6 & -5.06e-9 & -1.18e-11 & -1.65e-14 & -1.11e-17 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 
\end{bmatrix}$$

$$B_\delta = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

$$C_\delta = \begin{bmatrix} 5.00e-2 & 1.03e-3 & 8.84e-6 & 4.22e-8 & 1.18e-10 & 1.88e-13 & 1.38e-16 \end{bmatrix}$$

$$D_\delta = \begin{bmatrix} 13.0891 \end{bmatrix}$$

The transformation matrix $T$ of equation (4.25) is formulated and the systems of equation (4.26) formulated giving the following matrices:
The MATLAB function used to calculate the sensitivity matrices of the state space system is as given below

function [S_a,S_b,S_c] = sssens(a,b,c,d,Ts,operator);

% A function to calculate the sensitivity factors for a state space system.

% Ts = sampling period in seconds,
% operator = 'z' for Z operator formulation or 'd' for delta operator formulation.
% If a is for delta operator, it is assumed to be in the form:

% adelta = [1-d1 -d1 -d1 ... -d1 -d1
%           d2  1  0 ...  0  0
%           0  d3  1 ...  0  0
%           :  :  : ... :  :
%           0  0  0 ... d4  1]

% Written by: Dipesh

fs = 1 / Ts;

eigval = eig(a);           % eigval in Z

eigval = (1/Ts) * log(eigval);       % eigval in S

% Determine the frequency range

Architectural Considerations for a CSP
eigabs = abs(eigval);
minw = min(eigabs);
maxw = max(eigabs);

w = logspace(floor(log10(minw))-1, log10(maxw)+1, 1000);
w = w(:);
[row,col] = size(w);

order = size(a);
i_mat = eye(size(a));

[numd,dend] = ss2tf(a,b,c,d);
[magd,phased] = dbode(numd,dend,Ts,w);  % magnitude of original system

if operator(1) == 'z'
    z = exp(sqrt(-1) * w * Ts);
elseif operator(1) == 'd'
    z = exp(sqrt(-1) * w * Ts) - ones(row,col);
a = a - i_mat;
else
    error('Operator is unknown.');
end

z = z(:);
temp = 1;

% The following calculates the sensitivity factors at each frequency point
% for loop = 1 : row,
%  % dH/dB - equation (4.30)
%  % g_z(:,loop) = inv(z(loop) * i_mat - a') * c';
%
% dH/dC - equation (4.31)
% f_z(:,loop) = inv(z(loop) * i_mat - a) * b;
%
% dH/dA - equation (4.29)
% gf = inv(z(loop) * i_mat - a') * c' * b' * inv(z(loop) * i_mat - a');
for loop1 = 1 : order(1)
    % 2nd Term in equation (4.28)

Architectural Considerations for a CSP
\[ g_{z1}(\text{loop1,loop}) = |(g_z(\text{loop1,loop})) \cdot |b(\text{loop1})| / \text{magd(loop)}; \]

\%
3rd Term in equation (4.28)
%
\[ f_{z1}(\text{loop1,loop}) = |(f_z(\text{loop1,loop})) \cdot |c(\text{loop1})| / \text{magd(loop)}; \]

\%
1st Term in equation (4.28)
%
for loop2 = 1: order(2)
\[ g_{f_z}(\text{loop,temp}) = |(g_f(\text{loop1,loop2})) \cdot |a(\text{loop1,loop2})| / \text{magd(loop)}; \]
\[ \text{temp} = \text{temp} + 1; \]
end \% of loop2
end \% of loop1
\[ \text{temp} = 1; \]
end \% of loop

\%
Now to calculate the r.m.s value over all frequencies
%
\[ \text{temp} = 1; \]
for loop1 = 1: order(1)
if b(\text{loop1}) == 1
\[ S_b(\text{loop1}) = 0; \]
else
\[ S_b(\text{loop1}) = \text{norm}(g_{z1}(\text{loop1,:}),2) / \sqrt{\text{row}}; \]
end

if c(\text{loop1}) == 1
\[ S_c(\text{loop1}) = 0; \]
else
\[ S_c(\text{loop1}) = \text{norm}(f_{z1}(\text{loop1,:}),2) / \sqrt{\text{row}}; \]
end

for loop2 = 1: order(2)
if a(\text{loop1,loop2}) == 1,
\[ S_a(\text{loop1,loop2}) = \text{norm}(g_{f_z}(\text{,temp}),2) / \sqrt{\text{row}}; \]
else
\[ S_a(\text{loop1,loop2}) = 0; \]
end
\[ \text{temp} = \text{temp} + 1; \]
end
end

Architectural Considerations for a CSP
Appendix H

**PSEUDO-CODE FOR 2ND ORDER MODIFIED CANONIC STRUCTURE**

To implement the second order modified canonic structure shown in Figure 3.2, the following equations need to be implemented in software:

\[
\begin{align*}
v &= u - w - x \\
y &= p_1 v + p_2 w + p_3 x \\
x &= x + d_1 w \\
w &= w + d_1 v
\end{align*}
\]

The flow is:

- read input (u), \hspace{1cm} // Accumulator contains u
- calculate v, \hspace{1cm} // For Later use
- store v, \hspace{1cm} // For Later use
- calculate y, 
- output y, 
- calculate x, 
- store x, \hspace{1cm} // For Later use
- calculate w 
- store w \hspace{1cm} // For Later use

The above flow can be translated into pseudo-code as:
Pseudo-Code for 2nd Order Modified Canonic Structure

| read u | ; from the ADC |
| sub w | ; assuming w stored in register 'w' |
| sub x | ; assuming x stored in register 'x' |
| store v | ; in register 'v' |
| clear acc | ; clear accumulator |
| multacc p, v | ; coefficient p in register 'p' |
| multacc q, w | ; Acc = Acc + q*w |
| multacc r, x | ; Acc = Acc + q*w |
| output y | ; to DAC |
| clear acc | ; |
| load x | |
| multacc d2, w | ; x = x + d2, w |
| store x | |
| load w | |
| multacc d1, v | ; w = w + d1, v |

From the above certain instructions are common and will be needed regardless of the structure being implemented. These are:

- **load** (register value to accumulator)
- **store** (accumulator value to register)
- **multacc** (multiply and accumulate two operand and add result to value in accumulator. Result in accumulator).
- **subtract** (register value from accumulator value; result in accumulator)
- **add** (register value to accumulator value; result in accumulator)
- **read input** (from ADC)
- **write/output** (to DAC)
- **clear** (accumulator value)

Additionally, it has been assumed that the coefficients will be stored in on-chip registers. Therefore, instructions will be needed to allow for this. Other instructions that will be required are:

- **read/writes** (in case the on-chip register storage is not adequate)
- **to/from memory** (to an address location - thereby enabling looping)
- **jump** (from an interrupt service condition to normal mode and hence continue executing rest of program)

This forms the basis for the instruction set derived in section 5.3 and displayed in Table 5.2.
Appendix I

VHDL CODE FOR CSP MODEL

Package Description
-- A description of a package for the CSP model.
-- Written By : Dipesh I. Patel.
-- Last modified : 19th October 1994

library ieee, dzx;
use ieee.std_logic_1164.all;
use dzx.logic_utils.all;

package csp_types is
constant unit_delay : Time := 1ns;
type bool_to_bit_table is array(boolean) of bit;
constant bool_to_bit : bool_to_bit_table;
subsubtype bit_24 is bit_vector(23 downto 0);
subsubtype bit_16 is bit_vector(15 downto 0);
subsubtype bit_12 is bit_vector(11 downto 0);
subsubtype bit_11 is bit_vector(10 downto 0);
subsubtype bit_10 is bit_vector(9 downto 0);
subsubtype bit_5 is bit_vector(4 downto 0);
subsubtype bit_4 is bit_vector(3 downto 0);

-- Now a resolved type for Data Bus
subsubtype D_bus_bit_24 is X01Z_vector(23 downto 0);
subsubtype A_bus_bit_11 is X01Z_vector(10 downto 0);
constant op_nop : bit_5 := B"000000";
constant op_sub : bit_5 := B"000001";
constant op_add : bit_5 := B"000010";
constant op_mac : bit_5 := B"000111";
constant op_ldv : bit_5 := B"001000";
constant op_ldm : bit_5 := B"001010";
constant op_ldi : bit_5 := B"001011";
constant op_stv : bit_5 := B"010001";
constant op_stm : bit_5 := B"010100";
constant op_stc : bit_5 := B"010011";
constant op_clr : bit_5 := B"010100";

Architectural Considerations for a CSP
constant op JMP : bit_5 := B'010111';
constant op RTI : bit_5 := B'011000';
constant op RAD : bit_5 := B'011011';
constant op WDA : bit_5 := B'011100';
constant op ENI : bit_5 := B'011111';

function bits_to_int (bits : in bit_vector) return integer;
function bits_to_natural (bits : in bit_vector) return natural;
procedure int_to_bits (int : in integer; bits : out bit_vector);

end csp_types;

**Package Body**

```
-- A description of a package body for the CSP model.

-- Written By : Dipesh L. Patel.
-- Last modified : 19th October 1994

package body csp_types is

    constant bool_to_bit : bool_to_bit_table :=
        (false => '0', true => '1');

    function bits_to_int (bits : in bit_vector) return integer is
        variable temp : bit_vector(bits'range);
        variable result : integer := 0;
        begin
            if bits(bits'left) = '1' then -- negative number
                temp := not bits;
            else
                temp := bits;
            end if;
            for index in bits'range loop -- sign bit of temp = '0'
                result := result * 2 + bit'pos(temp(index));
            end loop;
            if bits(bits'left) = '1' then
                result := (-result) - 1;
            end if;
            return result;
        end bits_to_int;

    procedure int_to_bits(int : in integer; bits : out bit_vector) is
        variable temp : integer;
        variable result : bit_vector(bits'range);
        begin
            if int < 0 then
                temp := -(int + 1);
            else
                ... (remaining code not shown) ...
            end if;
        end int_to_bits;
```

Architectural Considerations for a CSP
temp := int;
end if;
for index in bits.reverse_range loop
    result(index) := bit'val(temp rem 2);
    temp := temp / 2;
end loop;
if int < 0 then
    result := not result;
    result(bits'left) := '1';
end if;
bis := result;
end int_to_bits;

function bits_to_natural (bits : in bit_vector) return natural is
    variable result : natural := 0;
    begin
    for index in bits.range loop
        result := (result * 2) + bit'pos(bits(index));
    end loop;
    return result;
end bits_to_natural;
end csp_types; -- end of package body

Architectural Considerations for a CSP
**Entity Description of Test Bench**

-- Description of the structure of the test bench circuit for CSP  --

-- Written By : Dipesh I. Patel  --
-- Last modified : 17th February 1995 --

-- This is the bench test circuit for the CSP. --

library work, dzx, ieee, std;

use work.csp_types.all;
use std.dazix_standard.all;
use dzx.logic_utils.X01Z_vector;
use ieee.std_logic_1164.X01Z;

define csp_test is

    port (phi1_out : out bit;
        phi2_out : out bit;
        reset_bar_out : out bit;
        int_bar_out : out bit;
        ios_bar_out : out bit;
        data_bar_out : out bit;
        read_bar_out : out bit;
        write_bar_out : out bit;
        prog_bar_out : out bit;
        en_int_out : out bit;
        ready_bar_out : out bit;
        d_bus_out : out X01Z_vector(23 downto 0);
        a_bus_out : out X01Z_vector(10 downto 0));

end csp_test;
VHDL Code for CSP Model

Structural Description of Test Bench

architecture structure of csp_test is

component clock_gen
    generic (Tpw : Time; Tps : Time);
    port (phi1, phi2, reset_bar : out bit);
end component;

component csp_ram
    generic (Tpd : Time := unit_delay);
    port (cs_bar, read_bar, write_bar, ready_bar, d_bus, a_bus : out bit);
end component;

component csp_eprom
    generic (Tpd : Time := unit_delay);
    port (cs_bar, read_bar, reset_bar, readY_bar, d_bus, a_bus : out bit);
end component;

component ADC_System
    generic (Tsamp : Time; Tcp, Tpw : Time; Tpd : Time := unit_delay);
    port (cs_bar, read_bar, reset_bar, int_bar, ready_bar, d_bus : out bit);
end component;

component DAC_System
    generic (Tpd : Time := unit_delay);
    port (cs_bar, write_bar, reset_bar, ready_bar, d_bus : out bit);
end component;

Architectural Considerations for a CSP
end component;

component CSP

generic (Tpd : Time := unit_delay);

port (  d_bus  : inout d_bus_bit_24 bus;
        a_bus  : out a_bus_bit_11 bus;
        read_bar  : out bit;
        write_bar  : out bit;
        prog_bar  : out bit;
        data_bar  : out bit;
        ios_bar  : out bit;
        en_int_bar  : out bit;
        int_bar  : in bit;
        reset_bar  : in bit;
        ready_bar  : in Dbit;
        phi1  : in bit;
        phi2  : in bit);

end component;
signal test_d_bus : D_bus_bit_24 bus;
signal test_a_bus : A_bus_bit_11 bus;
signal test_phi1, test_phi2 : bit;
signal rd_bar, wr_bar : bit;
signal rst_bar, test_int_bar, test_en_int_bar : bit;
signal int_act_bar : bit;
signal test_prog_bar, test_data_bar, test_ios_bar : bit;
signal rdy_bar : Dbit;

begin

c_gen : clock_gen generic map(8ns, 2ns)
    port map (phi1 => test_phi1,
              phi2 => test_phi2,
              reset_bar => rst_bar);

processor : csp generic map (1 ns)
    port map (d_bus => test_d_bus,
              a_bus => test_a_bus,
              read_bar => rd_bar,
              write_bar => wr_bar,
              prog_bar => test_prog_bar,
              data_bar => test_data_bar,
              ios_bar => test_ios_bar,
              en_int_bar => test_en_int_bar,
              int_bar => int_act_bar,
              reset_bar => rst_bar,
              ready_bar => rdy_bar,
              phi1 => test_phi1,
              phi2 => test_phi2);

test_ram : csp_ram generic map (1 ns)
    port map (cs_bar => test_data_bar,
              read_bar => rd_bar,
              write_bar => wr_bar,
              ready_bar => rdy_bar,
              d_bus => test_d_bus,
              a_bus => test_a_bus);
test_eprom : csp_eprom generic map (1 ns)
  port map (cs_bar => test_prog_bar,
            read_bar => rd_bar,
            reset_bar => rst_bar,
            ready_bar => rdy_bar,
            d_bus => test_d_bus,
            a_bus => test_a_bus);

test_ADC : ADC_System generic map (1.5us, 20ns, 8ns, unit_delay)
  port map (cs_bar => test_ios_bar,
            read_bar => rd_bar,
            reset_bar => rst_bar,
            int_bar => test_int_bar,
            ready_bar => rdy_bar,
            d_bus => test_d_bus);

test_DAC : DAC_System generic map (1 ns)
  port map (cs_bar => test_ios_bar,
            write_bar => wr_bar,
            reset_bar => rst_bar,
            ready_bar => rdy_bar,
            d_bus => test_d_bus);

int_act_bar <= test_int_bar or test_en_int_bar;
phi1_out <= test_phi1;
phi2_out <= test_phi2;
reset_bar_out <= rst_bar;
int_bar_out <= test_int_bar;
ios_bar_out <= test_ios_bar;
data_bar_out <= test_data_bar;
read_bar_out <= rd_bar;
write_bar_out <= wr_bar;
ready_bar_out <= rdy_bar;
prog_bar_out <= test_prog_bar;
en_int_out <= test_en_int_bar;
d_bus_out <= test_d_bus;
a_bus_out <= test_a_bus;
end structure;
Configuration file
-- ************************************************************ --
-- A description of configuration file for test bench circuit for CSP --
---written By : Dipesh I. Patel. --
-- Last modified : 16th February 1995 --
--
-- This is for the bench test circuit for the CSP. --
-- ******************************************************************** --

configuration csp_behaviour_test of csp_test is

  for structure
    for c_gen : clock_gen
      use entity work.clock_gen(behaviour);
    end for;

    for processor : csp
      use entity work.csp(behaviour);
    end for;

    for test_ram : csp_ram
      use entity work.csp_ram(behaviour);
    end for;

    for test_eprom : csp_eprom
      use entity work.csp_eprom(behaviour);
    end for;

    for test_ADC : ADC_System
      use entity work.ADC_System(behaviour);
    end for;

    for test_DAC : DAC_System
      use entity work.DAC_System(behaviour);
    end for;

  end for;

end csp_behaviour_test;
Entity Description of CSP

-- An entity description of the Control System Processor (CSP).

library ieee, work, std;
use std.dasix_standard.all;
use ieee.std_logic_1164.all;
use work.csp_types.all;

entity csp is

    generic (Tpd : Time := unit_delay);

    port ( d_bus : inout d_bus_bit_24 bus;
           a_bus : out a_bus_bit_11 bus;
           read_bar : out bit;
           write_bar : out bit;
           prog_bar : out bit;
           data_bar : out bit;
           ios_bar : out bit;
           en_int_bar : out bit;
           int_bar : in bit;
           reset_bar : in bit;
           ready_bar : in Dbit;
           phi1 : in bit;
           phi2 : in bit);

end csp;
Behavioral Description of CSP

-- A behavioural architecture description of the CSP.
-- Written By : Dipesh I. Patel.
-- Last modified : 23rd February 1995
-- This is one of the possible descriptions.

library ieee, work, dzx, std;
use std.dazix_standard.all;
use std.textio.all;
use ieee.std_logic_1164.all;
use dzx.logic_utils.all;
use work.csp_types.all;
use dzx.bit_arith.Shift_Right, dzx.bit_arith.To_Integer,
use dzx.bit_arith.Signed, dzx.bit_arith.To_Signed;

architecture behaviour of csp is

begin

architecture behaviour of csp is

subtype v_reg_addr is natural range 0 to 15;
subtype c_reg_addr is natural range 0 to 31;
subtype mem_addr is natural range 0 to 2048;
type v_reg_array is array (v_reg_addr) of bit_24;
type c_reg_array is array (c_reg_addr) of biL10;
begin

-- Variable Register File
-- Coefficient Register File
-- Program Counter - is 11 bit 'coz
-- it contains the address of the
-- next memory instruction

variable temp_PC: bit_24; -- temp storage for PC
variable MAR: biL11; -- Memory Address Register
variable IAR: bit_11; -- Interrupt Address Register
variable RIA: bit_11; -- Return from Interrupt Addr Reg
variable CI: bit_16; -- Current Instruction
variable Acc: bit_24; -- Accumulator
variable op: bit_5; -- Opcode Field
variable V_Addr: v_reg_addr; -- Address of Variable
variable C_Addr: c_reg_addr; -- Address of Coefficient
variable Immd_Val: bit_10; -- Immediate Value Field in Idi
variable Mem_Add: bit_11; -- Memory Address field in some inst.
variable dac_data: bit_24; -- output y to write to dac

constant z_state : X012_vector(23 downto 0) :=
procedure ROM_Read (address : in bit_11; result : out bit_16) is

-- This procedure is called whenever the processor is ready to read a new instruction.

begin

wait until phi2 = '0';
-- start bus cycle with address output (see timing figures to confirm)
  a_bus <= To_X01ZVector(address) after Tpd;
  prog_bar <= '0' after Tpd;

wait until phi1 = '1';
  leading edge of phi1 in T1 phase
if reset_bar = '0' then
  return;
end if;

-- T1 phase

read_bar <= '0' after Tpd;
wait until phi1 = '1';
  leading edge of phi1 in T2 phase
if reset_bar = '0' then
  return;
end if;

-- T2 phase

loop
  wait until phi2 = '0';
  if reset_bar = '0' then
    return;
  end if;
  -- end of T2
  if ready_bar = (F0) then
    result := To_BitVector(d_bus(15 downto 0));
    exit;
  end if;
end loop;

wait until phi1 = '1';
  leading edge of phi1 in T1 phase
if reset_bar = '0' then
  return;
end if;

-- T1 phase at end of cycle

read_bar <= '1' after Tpd;
  prog_bar <= '1' after Tpd;
end ROM_Read;
procedure RAM_Read (address : in bit_11; result : out bit_24) is

begin

wait until phi2 = '0';
-- start bus cycle with address output (see timing figures to confirm)
a_bus <= To_X01ZVector(address) after Tpd;
data_bar <= '0' after Tpd;

wait until phi1 = '1';   -- leading edge of phi1 in T1 phase
if reset_bar = '0' then
  return;
end if;

-- T1 phase

read_bar <= '0' after Tpd;
wait until phi1 = '1';   -- leading edge of phi1 in T2 phase
if reset_bar = '0' then
  return;
end if;

-- T2 phase

loop
  wait until phi2 = '0';
  if reset_bar = '0' then
    return;
  end if;
  -- end of T2
  if ready_bar = (F0) then
    result := To_BitVector(d_bus);
    exit;
  end if;
end loop;

wait until phi1 = '1';   -- leading edge of phi1 in Ti phase
if reset_bar = '0' then
  return;
end if;

-- Ti phase at end of cycle

read_bar <= '1' after Tpd;
data_bar <= '1' after Tpd;
end RAM_Read;
procedure Memory_Write (address : in biC11; data : in biC24) is

begin

  wait until phi2 = '0';
  -- start bus cycle with address output (see timing figures to confirm)
  a_bus <= To_X01ZVector(address) after Tpd;
  data_bar <= '0' after Tpd;
  prog_bar <= '1' after Tpd;

  wait until phi1 = '1';  -- leading edge of phi1 in T1 phase
  if reset_bar = '0' then
    return;
  end if;

  -- T1 phase

  write_bar <= '0' after Tpd;
  wait until phi2 = '1';  -- leading edge of phi2 in T1 phase
  d_bus <= To_X01ZVector(data) after Tpd;
  wait until phi1 = '1';  -- leading edge of phi1 in T2 phase
  if reset_bar = '0' then
    return;
  end if;

  -- T2 phase

  loop
    wait until phi2 = '0';
    if reset_bar = '0' then
      return;
    end if;

    -- end of T2
    exit when ready_bar = (FO);
  end loop;

  wait until phi1 = '1';  -- leading edge of phi1 in Ti phase
  if reset_bar = '0' then
    return;
  end if;

  -- Ti phase at end of cycle

  write_bar <= '1' after Tpd;
  data_bar <= '1' after Tpd;
  d_bus <= z_state after Tpd;

end Memory_Write;
procedure Read_ADC (result : out bit_12) is

-- This procedure is called whenever the processor is needs to read
-- data from the ADC.

begin

wait until phi2 = '0';
ios_bar <= '0' after Tpd;

wait until phi1 = '1';
if reset_bar = '0' then
  return;
end if;

-- T1 phase
read_bar <= '0' after Tpd;
wait until phi1 = '1';
if reset_bar = '0' then
  return;
end if;

-- T2 phase
loop
  wait until phi2 = '0';
  if reset_bar = '0' then
    return;
  end if;
  if ready_bar = (F0) then
    result := To_BitVector(d_bus(23 downto 12));
    exit;
  end if;
end loop;

wait until phi1 = '1';
if reset_bar = '0' then
  return;
end if;

-- Ti phase at end of cycle
read_bar <= '1' after Tpd;
ios_bar <= '1' after Tpd;
end Read_ADC;
procedure Write_DAC (dac_data : in bit_24) is

-- This procedure is called whenever the processor is needs to write data to the DAC.

begin

    wait until phi2 = '0';
    ios_bar <= '0';

    wait until phi1 = '1';
    if reset_bar = '0' then
        return;
    end if;

begin

    write_bar <= '0' after Tpd;
    wait until phi2 = '1';

    d_bus <= To_X01ZVector(dac_data) after Tpd;

    wait until phi1 = '1';
    if reset_bar = '0' then
        return;
    end if;

loop

wait until phi2 = '0';
    if reset_bar = '0' then
        return;
    end if;

exit when ready_bar = (FO);
end loop;

wait until phi1 = '1';
    if reset_bar = '0' then
        return;
    end if;

-- Ti phase at end of cycle

write_bar <= '1' after Tpd;
    ios_bar <= '1' after Tpd;
    d_bus <= z_state after Tpd;
end Write_DAC;
procedure CSP_Add (op1, op2 : in integer; result : out bit_24) is

  -- The following procedure Add is used to add 2 24-bit two's complement numbers together.

begin
  int_to_bits(op1 + op2, result);
end CSP_Add;

procedure Subtract (op1, op2 : in bit_24; result : out bit_24) is

  -- The following procedure is used to subtract 2 24-bit two's complement numbers from each other.

begin
  int_to_bits(op1, result);
end Subtract;

procedure MultAcc (op1, op2 : in integer; coeff : in bit_vector; result : out bit_24) is

  -- The following Multiply a 24 bit 2's complement number with an unsigd number and accumulate the result with the accumulator value.

begin
  -- op1 = accumulator value
  -- op2 = variable
  -- result = op1 - op2
  temp1 := bits_to_int(op1);
  temp2 := bits_to_int(op2);
  int_to_bits(temp1 - temp2, result);
end Subtract;

Architectural Considerations for a CSP
begin

    -- First Split the Coef into significant value and number of shift
    -- bits

    SigVal := bits_to_natural(coeff(5 downto 0));
    ExpBits := coeff(9 downto 6);
    ExpVal := bits_to_natural(ExpBits);

    temp1 := op2 * SigVal;
    temp2 := To_Signed(temp1, 30);
    temp2 := Shift_Right(temp2, (ExpVal + 6)); -- align dec pnt due to coeff

    temp1 := To_Integer(temp2);
    csp_add(temp1, op1, result);

end MultAcc;

begin -- of the process statement

    -- check for reset_bar active
    if reset_bar = '0' then
        read_bar <= '1' after Tpd;
        write_bar <= '1' after Tpd;
        ios_bar <= '1' after Tpd;
        data_bar <= '1' after Tpd;
        prog_bar <= '1' after Tpd;
        d_bus <= z_state after Tpd;
        en_int_bar <= '1' after Tpd;
        PC := B'000_0000_0000'; -- Initialise PC to 0
        RIA := B'000_0000_0000';
        MAR := B'000_0000_0000';
        IAR := B'000_0000_0010';
        Acc := X'00_0000'; -- Initialise Acc to 0
        wait until reset_bar = '1';
    end if;

    -- check for interrupt (int bar) active
    if reset_bar /= '0' and int_bar = '0' then
        RIA := PC; -- save PC value in RIA
        PC := IAR; -- load PC with ISR address

        -- wait for int_bar to become inactive before proceeding
        --
        wait until int_bar = '1';
    end if;

    -- fetch next instruction
    --
    ROM_read(PC, Cl);

    if reset_bar /= '0' and int_bar /= '0' then

Architectural Considerations for a CSP

215
csp_add(bits_to_int(PC),1,temp_PC);
PC := temp_PC(10 downto 0);

-- decode and execute
--

op := Cl(15 downto 11);
Mem_Add := Cl(10 downto 0);
Immd_Val := Cl(10 downto 1);
C_Addr := bits_to_natural(CI(10 downto 6));
V_Addr := bits_to_natural(CI(5 downto 2));

case op is
when op_add =>
    CSP_Add(bits_to_int(Acc), bits_to_int(v_reg(V_Addr)), Acc);
when op_sub =>
    Subtract(Acc, v_reg(V_Addr), Acc);
when op_mac =>
    MulAcc(bits_to_int(Acc), bits_to_int(v_reg(V_Addr)),
           c_reg(C_Addr), Acc);
when op_clr =>
    Acc := X"00_0000";
when op_ldm =>
    if reset_bar /= '0' then
        RAM_read(Mem_Add, Acc);
    end if;
when op_ldv =>
    Acc := v_reg(V_Addr);
when op_ldi =>
    Acc(9 downto 0) := Immd_Val;
    Acc(23 downto 10) := B"00_0000_0000_0000";
when op_stm =>
    if reset_bar /= '0' then
        Memory_Write(Mem_Add, Acc);
    end if;
when op_stv =>
    v_reg(V_Addr) := Acc;
when op_stc =>
    c_reg(C_Addr) := Acc(9 downto 0);
when op_imp =>
    if reset_bar /= '0' or int_bar /= '0' then
        PC := Mem_Add;
    end if;
when op_NOP =>
    if reset_bar /= '0' or int_bar /= '0' then
        PC := PC;
    end if;
when op_rti =>
    PC := RIA;
when op_rad =>
  if reset_bar /= '0' then
    Read_ADC(Acc(23 downto 12));
    Acc(11 downto 0) := B"0000_0000_0000";
  end if;

when op_wda =>
  if reset_bar /= '0' then
    --
    dac_data(23 downto 12) := Acc(23 downto 12);
    --
    dac_data(11 downto 0) := X"000";
    dac_data := Acc;
    Write_DAC(dac_data);
    out_data := bits_to_int(Acc);
    write(out_line, out_data);
    writeln(outfile, out_line);
  end if;

when op_eni =>
  en_int_bar <= '0';

when others =>
  assert false report "illegal instruction" severity warning;

end case;

end if; -- reset_bar /= '0' and int_bar /= 0

end process;

end behaviour;
Clock Generator Description

.. **Clock Generator Description**

-- A description of a clock generator module.

-- Written By : Dipesh I. Patel.

-- Last modified : 16th February 1995

-- This will be used in the bench test circuit for the CSP.

library ieee, work;
use ieee.std_logic_1164.all;
use work.csp_types.all;

entity clock_gen is

    generic (Tpw : Time := 8 ns; Tps : Time := 2 ns);

    port (phi1 : out bit;

          phi2 : out bit;

          reset_bar : out bit);

end clock_gen;

architecture behaviour of clock_gen is

    constant clock_period : Time := 2*(Tpw + Tps);

begin

    reset_driver :

        reset_bar <= '0', '1' after 2 * clock_period + Tpw;

    clock_driver : process

        begin

            phi1 <= '1', '0' after Tpw;

            phi2 <= '1' after Tpw + Tps, '0' after Tpw + Tps + Tpw;

            wait for clock_period;

        end process clock_driver;

end behaviour;
ADC Subsystem Description

-- **************************************************************************--
-- A description of a ADC subsystem module.                             --
--                                                                       --
-- Written By : Dipesh I. Patel.                                      --
-- Last modified : 22nd February 1995                                 --
-- This will be used in the bench test circuit for the CSP.            --
-- **************************************************************************--

library ieee, work, dzx, std;
use ieee.std_logic_1164.all;
use std.dazix_standard.all;
use work.csp_types.all;
use dzx.logic_utils.all;

entity ADC_System is
generic (Tsamp : Time := 1 us; Tcp : Time := 20ns; Tpw : Time := 8ns;
         Tpd : Time := unit_delay);
port (cs_bar : in bit;
     read_bar : in bit;
     reset_bar : in bit;
     int_bar : out bit;
     ready_bar : out Dbit;
     d_bus : out X01Z_vector(23 downto 0));
end ADC_System;

architecture behaviour of ADC_System is

   constant sampling_period : Time := Tsamp;
   constant clock_period : Time := Tcp;

   -- ADC_Output:
   -- output = 001000 implies unit step input to controller
   -- output = 0FF000 implies high positive input
   -- output = FFF000 implies negative input of -1
   constant ADC_Output : bit_vector(23 downto 0) := X"FFFOOO";
   constant z_state : X01Z_vector(23 downto 0) :=
     ('Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z');

begin
   Interrupt_Generate : process
      -- Generates an Interrupt signal which is held low for 3.5
      -- clock periods every Tsamp secs
      begin
         wait until reset_bar = '1';

         loop
            int_bar <= '0', '1' after 3 * clock_period + Tpw;
            wait for sampling_period;
         end loop;
   end process;
end behaviour;
end process Interrupt_Generate;

Data_Write : process  
begin
    -- put d_bus and ready into an initial state
    --
    ready_bar <= (Z1) after Tpd;
    d_bus <= z_state after Tpd;
    --
    -- wait for a command
    --
    wait until (cs_bar = '0' and read_bar = '0');
    --
    -- Processor ready to read
    --

    d_bus <= To_X01ZVector(ADC_Output) after Tpd;
    ready_bar <= (F0);

    wait until (read_bar = '1' or cs_bar = '1');

end process Data_Write;

end behaviour;
DAC Subsystem Description

-- ****************************************************************** --
-- A description of a DAC subsystem module.                   --
-- Written By : Dipesh I. Patel.                           --
-- Last modified : 19th February 1995                     --
-- This will be used in the bench test circuit for the CSP. --
-- ****************************************************************** --

library ieee, work, dzx, std;

use std.dazix_standard.all;
use ieee.std_logic_1164.all;
use work.csp_types.all;
use dzx.logic_utils.all;

entity DAC_System is
  generic (Tpd : time := unit_delay);
  port (cs_bar : in bit;
         write_bar : in bit;
         reset_bar : in bit;
         ready_bar : out bit;
         d_bus : in X01Z_vector (23 downto 0));
end DAC_System;

architecture behaviour of DAC_System is

  subtype dac_space is natural range 0 to 5000;
  type dac_array is array (dac_space) of bit_24;

begin

  Read_Data : process

    variable dac_data : dac_array;
    variable dac_index : natural;
    constant high : bit := '1';
    constant low : bit := '0';

    begin

     -- Initialises the dac_data array to 0 on reset
      if reset_bar = '0' then
        for i in 0 TO 5000 loop
          dac_data(i) := X"0000000";
        end loop;

        dac_index := 0;

        wait until reset_bar = '1';

      end if;

    end process Read_Data;

Architectural Considerations for a CSP
-- ready into an initial state
ready_bar <= (Z\_1) after Tpd;

-- wait for a command
wait until (cs\_bar = '0' and write\_bar = '0');

-- data is valid; perform read from d\_bus
dac\_data(dac\_index) := To\_BitVector(d\_bus);
ready_bar <= (F0) after Tpd;
wait until (write\_bar = '1' or cs\_bar = '1'); -- hold
dac\_index := dac\_index + 1; -- point to next value

end process Read\_Data;

end behaviour;
**EPROM Description**

-- **************************************************************************
-- | A description of an EPROM module.             |
-- | Written By : Dipesh I. Patel.               |
-- | Last modified : 19th February 1995          |
-- | This will be used in the bench test circuit for the CSP. |
-- **************************************************************************

library ieee, work, std, dzx;
use std.dazix_standard.all;
use ieee.std_logic_1164.all;
use work.csp_types.all;
use dzx.logic_utils.all;

entity csp_eprom is
  generic (Tpd : Time := uniCdelay);
  port (cs_bar : in bit;
        read_bar : in bit;
        reset_bar : in bit;
        ready_bar : out Dbit;
        d_bus : out X01Z_vector(23 downto 0)
        bus;
        a_bus : in A_bus_bit_11 bus);
end csp_eprom;

architecture behaviour of csp_eprom is
begin
  process
    constant low_address : integer := 0;
    constant high_address : integer := 2048;
    constant z_state : X01Z_vector(23 downto 0) :=
      ('Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z',
       'Z','Z','Z','Z','Z','Z','Z','Z','Z','Z');
    type memory_array is array (integer range low_address to high_address) of bit_24;
    variable mem : memory_array;
    variable address : integer;

    begin
      -- Initialise Memory values (mimick program in EPROM).
      -- Only initialises when reset is active (low)
      if reset_bar = '0' then
        for i in 0 TO 2048 loop
          mem(i) := x"000000";
        end loop;
      end if;
  end process;

end behaviour;
These are the values that are given in Appendix J - The software that implements the Phase Advance Controller.

Architectural Considerations for a CSP

```vhdl
mem(0) := X'005810';
mem(2) := X'005820';
mem(16) := X'0031E0';
mem(17) := X'004800';
mem(18) := X'005000';
mem(19) := X'003800';
mem(20) := X'003804';
mem(21) := X'007800';
mem(22) := X'000000';
mem(23) := X'005816';
mem(32) := X'006800';
mem(33) := X'000804';
mem(34) := X'003800';
mem(35) := X'001000';
mem(36) := X'001000';
mem(37) := X'001000';
mem(38) := X'001000';
mem(39) := X'001000';
mem(40) := X'007000';
mem(41) := X'005000';
mem(42) := X'002004';
mem(43) := X'001800';
mem(44) := X'003804';
mem(45) := X'006000';

wait until reset_bar = '1';

end if;

--
-- put d_bus and ready into an initial state
--
d_bus <= z_state after Tpd;
ready_bar <= (Z1) after Tpd;

--
-- wait for a command
--
wait until cs_bar = '0';

--
-- dispatch read or write cycle
--
address := bits_to_int(To_BitVector(a_bus));
if address >= low_address and address <= high_address then
    -- Address match for this memory
    d_bus <= To_X01ZVector(mem(address)) after Tpd;
    ready_bar <= (F0) after Tpd;
    wait until read_bar = '1';
    -- hold for read cycle
end if;

end process;

end behaviour;
```
**RAM Description**

-- A description of a RAM module.

-- Written By : Dipesh I. Patel.
-- Last modified : 17th February 1995

-- This will be used in the bench test circuit for the CSP.

library ieee, work, dzx, std;
use std.dazix_standard.all;
use ieee.std_logic_1164.all;
use work.csp_types.all;
use dzx.logic_utils.all;

entity csp_ram is
  generic (Tpd : Time := unit_delay);
  port (cs_bar : in bit;
        read_bar : in bit;
        write_bar : in bit;
        ready_bar : out Obit;
        d_bus : inout D_bus_bit_24 bus;
        a_bus: in A_bus_bit_11 bus);
end csp_ram;

architecture behaviour of csp_ram is
begin

  process
  begin

    constant low_address: integer := 0;
    constant high_address: integer := 2048;
    constant z_state : X01Z_vector(23 downto 0) :=
      ( 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z',
        'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z' 'Z');
    type memory_array is array (integer range low_address to high_address) of bit_24;
    variable mem : memory_array;
    variable address: integer;

    begin

      -- put d_bus and ready into an initial state
      d_bus <= z_state after Tpd;
      ready_bar <= (Z1) after Tpd;

      -- wait for a command
      wait until cs_bar = '0';

      -- dispatch read or write cycle

  end process;

end;
address := bits_to_int(To_BitVector(a_bus));
if address >= low_address and address <= high_address then
   -- Address match for this memory
   if write_bar = '0' then
      ready_bar <= (F0) after Tpd;
      wait until write_bar = '1'; -- hold for write cycle
      -- Sample data from Tpd ago is now stored
      mem(address) := To_BitVector(d_bus'delayed(Tpd));
   else -- read_bar = '0'
      d_bus <= To_X01ZVector(mem(address)) after Tpd;
      ready_bar <= (F0);
      wait until read_bar = '1'; -- hold for read cycle
   end if;
end if;
end process;
end behaviour;
Appendix J

**PHASE ADVANCE CONTROLLER (ASSEMBLY & MACHINE CODE)**

The code to implement the phase advance controller on the CSP using the instruction set defined in Chapter 5 is given in this section. The assembly code is given first followed by the equivalent machine code along with the memory locations where each instruction will be stored in the EPROM.

**Assembly Code:**

<table>
<thead>
<tr>
<th>Label</th>
<th>Assembly Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>org 00H</td>
<td>; assembler directive to initiate at address 00H</td>
<td></td>
</tr>
<tr>
<td>jmp main</td>
<td>; jump to main program</td>
<td></td>
</tr>
<tr>
<td>org 02H</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>jmp isr</td>
<td>; jmp to interrupt service routine (isr)</td>
<td></td>
</tr>
<tr>
<td>org 10H</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>main ldi d1</td>
<td>; load immediate value into accumulator (coefficient $d_1$)</td>
<td></td>
</tr>
<tr>
<td>stc RC0</td>
<td>; store $d$, in C-Reg file address 0</td>
<td></td>
</tr>
<tr>
<td>clr acc</td>
<td>; clear accumulator</td>
<td></td>
</tr>
<tr>
<td>stv RV0</td>
<td>; initialise $v$ to zero</td>
<td></td>
</tr>
<tr>
<td>stv RV1</td>
<td>; initialise $w$ to zero</td>
<td></td>
</tr>
<tr>
<td>eni</td>
<td>; enable interrupt</td>
<td></td>
</tr>
<tr>
<td>loop nop</td>
<td>; wait for timing signal</td>
<td></td>
</tr>
<tr>
<td>jmp loop</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>org 20H</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>isr rad</td>
<td>; beginning of isr - read from ADC</td>
<td></td>
</tr>
<tr>
<td>sub RV1</td>
<td>; $v = u - w$</td>
<td></td>
</tr>
<tr>
<td>stv RV0</td>
<td>; store $v$ for later use</td>
<td></td>
</tr>
<tr>
<td>add RV0</td>
<td>; $v = v + v$ (repeated 5 time because $p = 5$)</td>
<td></td>
</tr>
<tr>
<td>add RV0</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>add RV0</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>add RV0</td>
<td>; p time $v$ ready now</td>
<td></td>
</tr>
<tr>
<td>add RV1</td>
<td>; $y = pv + qw$ ($q = 1$ for this particular case)</td>
<td></td>
</tr>
<tr>
<td>wda</td>
<td>; output $y$ to DAC</td>
<td></td>
</tr>
<tr>
<td>clr</td>
<td>; acc = 0</td>
<td></td>
</tr>
</tbody>
</table>

*Architectural Considerations for a CSP* 227
**Phase Advance Controller - Assembly & Machine Code**

```plaintext
ldv RV1  ; load w
mac RC0, RV0  ; w = w + d, v
stv RV1  ; store w for use during next sample
rti  
```

**Note:**

RV0 is variable register address 00H
RV1 is variable register address 01H
RC0 is coefficient register address 00H

In **machine code**, the above translates as:

<table>
<thead>
<tr>
<th>EPROM Memory Location</th>
<th>Machine Code (Binary - 16 bits)</th>
<th>Machine Code (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>org 00H</td>
<td>00H 01011 0000010000</td>
<td>5810H</td>
</tr>
<tr>
<td>jmp main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org 02H</td>
<td>02H 01011 0000010000</td>
<td>5820H</td>
</tr>
<tr>
<td>jmp isr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org 10H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldi d1</td>
<td>10H 00110 001111000000</td>
<td>31E0H</td>
</tr>
<tr>
<td>stc RC0</td>
<td>11H 01001 000000000000</td>
<td>4800H</td>
</tr>
<tr>
<td>clr acc</td>
<td>12H 01010 000000000000</td>
<td>5000H</td>
</tr>
<tr>
<td>stv RV0</td>
<td>13H 00111 000000000000</td>
<td>3800H</td>
</tr>
<tr>
<td>stv RV1</td>
<td>14H 00111 000000000100</td>
<td>3804H</td>
</tr>
<tr>
<td>eni</td>
<td>15H 01111 000000000000</td>
<td>7800H</td>
</tr>
<tr>
<td>NOP</td>
<td>16H 00000 000000000000</td>
<td>0000H</td>
</tr>
<tr>
<td>jmp loop</td>
<td>17H 01011 000001011000</td>
<td>5816H</td>
</tr>
<tr>
<td>org 20H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rad</td>
<td>20H 01101 000000000000</td>
<td>6800H</td>
</tr>
<tr>
<td>sub RV1</td>
<td>21H 00001 000000001000</td>
<td>0804H</td>
</tr>
<tr>
<td>stv RV0</td>
<td>22H 00111 000000000000</td>
<td>3800H</td>
</tr>
<tr>
<td>add RV0</td>
<td>23H 00010 000000000000</td>
<td>1000H</td>
</tr>
<tr>
<td>add RV0</td>
<td>24H 00010 000000000000</td>
<td>100H</td>
</tr>
<tr>
<td>add RV0</td>
<td>25H 00010 000000000000</td>
<td>100H</td>
</tr>
<tr>
<td>add RV0</td>
<td>26H 00010 000000000000</td>
<td>1000H</td>
</tr>
<tr>
<td>add RV1</td>
<td>27H 00010 000000001000</td>
<td>1004H</td>
</tr>
<tr>
<td>wda</td>
<td>28H 01110 000000000000</td>
<td>7000H</td>
</tr>
<tr>
<td>clr</td>
<td>29H 01010 000000000000</td>
<td>5000H</td>
</tr>
<tr>
<td>ldv RV1</td>
<td>2AH 00100 000000001000</td>
<td>2004H</td>
</tr>
<tr>
<td>mac RC0, RV0</td>
<td>2BH 00011 000000000000</td>
<td>1800H</td>
</tr>
<tr>
<td>stv RV1</td>
<td>2CH 00111 000000001000</td>
<td>3804H</td>
</tr>
<tr>
<td>rti</td>
<td>2DH 01100 000000000000</td>
<td>6000H</td>
</tr>
</tbody>
</table>
```

Architectural Considerations for a CSP
Close examination of the EPROM Model in the VHDL code (Appendix I) will show that the above binary/hex values appear in the corresponding memory locations. That is they have been 'blown' onto the EPROM.