ON THE DESIGN AND IMPLEMENTATION OF A CONTROL SYSTEM PROCESSOR

by

René Armando Cumplido Parra

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy
of
Loughborough University

2001

Abstract

In general digital control algorithms are multi-input multi-output (MIMO) recursive digital filters, but there are particular numerical requirements in control system processing for which standard processor devices are not well suited, in particular arising in systems with high sample rates. There is therefore a clear need to understand the numerical requirements properly, to identity optimised forms for implementing control laws, and to translate these into efficient processor architectures. By taking a considered view of the numerical and calculation requirements of control algorithms, it is possible to consider special purpose processors that provide well-targeted support of control laws.

This thesis describes a compact, high-speed, special-purpose processor which offers a low-cost solution to implementing linear time invariant controllers. The overall approach involves re-formulating the controller into a particular discrete state-space representation, optimised for numerical efficiency using the δ operator, then programming this into a custom Control System Processor (CSP) implemented using a 'programmable ASIC' device.

The numerical optimisation means that the real-time processing is more accurate, thus the wordlength required to represent the variables and coefficients is reduced. These representations of coefficients and state variables are satisfactory for a wide-range of controllers. This novel architecture, which incorporates a targeted multiplier-accumulator (MAC) unit optimised for calculating the sum of products, combined with the use of a small and specialised instruction set, presents cost and performance benefits for control applications over traditional architectures. The CSP's dedicated architecture and careful numerical formulation ensure that it will perform deterministically in a real-time embedded control environment.

The design of a simplified hardware multiply-and-accumulate unit resulted in a high-speed, low power, low cost numerically stable processor for embedded control. A comprehensive set of tests has shown that the CSP operates correctly on a variety of filter types over a range of input conditions. The control system processor was successfully implemented and verified on a programmable device. The results of a benchmark indicate that the control system processor outperforms some commercially available high-speed processors by a significant margin when implementing the example controllers.

The CSP is a compact, high-speed special purpose processor, which enables a low-cost solution to a wide range of LTI control problems. It offers a very effective implementation for embedded control and it is applicable to any solution of IIR filters. The modest gate count of the CSP confers a number of advantages, namely reduced cost due to small die size and simpler packaging, and low power.

Acknowledgements

I wish to thank my supervisors Professors Simon Jones and Roger Goodall, for their support and guidance during my research. I also thank Professor Steve Bateman for his help and encouragement. I have been fortunate to have the opportunity to work with and learn from them. I owe them my deepest gratitude.

For my financial support I thank the National Council for Science and Technology of México (CONACyT), without whom this work would have not been possible.

I sincerely thank all members of the Electronic Systems Design Group at Loughborough University, both past and present, for their friendship and for all those interesting discussions at lunchtime.

By far the most important support came from my family. I thank my parents, René and Elba, for the education and encouragement I have always received at home. I also thank my wife, Claudia, for her love and support throughout these years. Finally, I must thank the rest of my family and friends. I am much indebted to all of them.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 INTRODUCTION

Rapid advances in electronics especially in the techniques used to manufacture integrated circuits, parallel to the development of new techniques and methodologies of modern control theory, have already had, and will continue to have a major impact on a number of industrial disciplines and applications. The need to provide cost-effective implementation of control systems becomes evident especially in high-performance electro-mechanical applications. Some examples can be found in industrial drives, automotive and aerospace control, where controllers are usually embedded into the system. Embedded real-time control is a particularly demanding application domain since the calculations must often be performed to meet hard time deadlines. To satisfy the demands of these applications, control systems must calculate complex recursive digital filters in real time.

This thesis investigates a method and processor architecture for the construction of high-performance processors targeted at linear time-invariant (LTI) control. The overall methodology involves re-formulating the controller formulation into a particular discrete state-space representation, which is optimised for numerical efficiency, then programming this into a specially-designed Control System Processor (CSP) implemented using a 'programmable ASIC' device.

1.2 MOTIVATION

Digital control systems are characterised by the algorithms used. The algorithms specify the arithmetic operations to be performed but do not specify how that arithmetic is to be implemented. The selection of a specific technology is affected in part by the required speed and arithmetic factors derived for the control algorithm, resulting in a variety of different combinations of algorithms, hardware, and software. The availability of control processing solutions, which are both efficient and straightforward to use, are key elements for the achievement of robust and cost-effective solutions.

The availability of cheap and powerful digital computing together with powerful tools for analysis, design and simulation, have dramatically transformed control engineering, with digital processors now used to perform a wide range of roles both in embedded processing and supervisory control [Irwin98].

As commercially available high-speed general-purpose processors have become faster and more complex, they have increased the competitiveness of digital implementations of control systems. Some processor architectures include a number of additional features that facilitate the implementation of digital control. Also, these processors allow new features to be added to an existing control system by modifying only the software that implements the control algorithm. This is possible due to the multitude of functions these processors are designed to perform and the powerful software development tools that take full advantage of those features.

All the advantages of using general-purpose processors come with a cost. The control algorithm has to be artificially partitioned and constrained to meet the physical bus widths and mapped on the instruction set. Furthermore, any parallelism inherent in the algorithm will be lost when it is translated into the serial code performed by the processor. Operations such as multiplication, that are essential to perform digital control, are usually decomposed into a sequence of simpler operations performed by numerical routines. Additionally, the flexibility provided by the processor is not needed to implement many digital control applications, and for any given clock cycle, only a small portion of the logic elements on the device may be doing something associated with the control process. Furthermore, the execution time for control system software can be difficult to predict due to software complexity and resources like cache memory, pipelining, interruptions, etc. Other restrictions on the controller device such as power consumption and cost might prove to be difficult to overcome, and as a consequence these processors are often not considered when implementing low-cost systems.

Real-time operation is critical in control applications, this means that even if a correct result is obtained, it will be useless if it appears too late. Thus, it is required that the controller processes data at a speed that is closely related to the sample rate of the system inputs. Also, because the loss of real-time operation is not tolerated, the controller behaviour must be predictable.

Program control in digital controllers must be oriented towards fast execution of loops of code. While zero-overhead looping is a desired feature, branching is rarely needed [Martin98]. Precision and data types vary and compact data storage is important; normal byte boundaries may prove inadequate for some applications with special requirements of precision and dynamic range [Goodall92].

In almost any application special-purpose processors provide better performance than programmable processors due to their specialised nature. The possibility of designing a special-purpose processor for control systems offers potential benefits. It will substantially increase the processing capacity and reduce the size of the system and consequently power consumption. Additionally, by exploring the computational properties of the algorithm to be implemented and designing architectures that match the algorithm and not vice versa, special-purpose processors offer a reasonable approach when implementing applications with the high sampling rates needed in high-performance closed-loop control systems. Furthermore, for high-volume products, special-purpose processors may also be less expensive as only those functions needed by the application are implemented.

Advances in system design capabilities and semiconductor technology have made possible to economically design custom architectures, so new innovative systems solutions for use in dedicated applications such as digital control can be explored. Modern programmable devices, such as Field Programmable Gate Arrays (FPGAs), offer many advantages when used as a prototype in systems design. They also offer a low cost alternative for fast experimental work and give the designer the opportunity to modify the design at the modelling and hardware stages. Efficient real-time implementation requires attention to both algorithm and architecture, and also a combination of control engineering and electronic system design skills

Implementations using special-purpose processors and general-purpose processors are not mutually exclusive. In fact, they may be integrated to provide a more complex solution where the special-purpose processor performs the computationally demanding operations and the general-purpose processor performs the additional functions needed to implement real-time digital control.

1.3 Objectives of the research

The aim of this research is to investigate whether by providing customised hardware support for control we can provide a low cost, high-performance embedded controller. The system must be efficient (low complexity and high speed), capable of handling most types of advanced LTI controllers and perform deterministically in a real-time embedded control environment. Also, it is desired that the system minimises computational delay and has large dynamic range.

Digital control can be seen as the arithmetic processing of signals sampled at regular intervals to obtain desired signals at the output [Nekoogar99]. These arithmetic operations are dictated by the control algorithm, thus, to implement an efficient digital controller we must understand the basic operations and functions contained within the control algorithms.

By analysing strengths and shortcomings of current digital controllers and the system requirements, several issues regarding the type of architecture can be investigated. Some of most important issues are: general arithmetic architecture, instruction set, ability to perform arithmetic operations in one cycle, amount of storage space, bus architecture, and pipelining.

We must select some example controllers to show that the proposed architecture can satisfy a range of high sample rate controller and to prove the numerical aspects of its operation, and to produce a benchmark comparison that includes some of the most popular processors used for control. Also, a method to implement the controllers using the knowledge gained from this research has to be proposed. This includes the development of software utilities to support the implementation of different control algorithms.

1.4 STRUCTURE OF THE THESIS

Chapter 1 presents an introduction, motivations and objectives of this research work.

Chapter 2 provides a definition of a control system and presents a general brief review of the control theory concepts needed for the appreciation of this work. It includes a review of several processor devices used to implement digital controllers and highlights the potential benefits of using special-purpose architectures. It includes a section that describes software issues related to the implementation of digital control systems. Finally, a literature survey of related work is presented and analyses the direction to go.

Chapter 3 further explains the objectives of this work. It identifies the investigations to be carried out and details the methodology and tools used to fulfil the objectives. Also, it describes the environment in which the experiments are carried out and the assumptions taken to perform the experiments.

Chapter 4 presents the fundamentals of modern digital control systems design. It analyses the structure adopted to implement the control algorithm and highlight its advantages when compared to traditional approaches. Also, it identifies crucial aspects to be considered when implementing an efficient digital controller.

Chapter 5 explains the implementation and essential components of the CSP architecture. It explains the mapping methodology used to create the architecture based on the selected control formulation. A detailed description of the CSP core, including the special-purpose multiply-accumulator unit is given. Finally, it presents the results of synthesising the architecture in terms of speed and complexity.

Chapter 6 explains the software scheme adopted to create the CSP program. It describes the CSP instruction set and instruction format, and gives a brief description of the software suite that supports the CSP concept.

Chapter 7 presents the results of benchmarking the CSP against other processors. It describes the controller examples and processors used in the benchmark. It explains the assumptions taken for the benchmarks. Finally, it shows and analyses some simulation results.

Chapter 8 concludes this thesis and evaluates the results obtained in this work by discussing the strengths and shortcomings of the proposed architecture. Finally, a framework of potential future work is presented.

Chapter 2

Digital Control Issues and Literature Survey

2.1 OBJECTIVES OF THE CHAPTER

This chapter presents a review of the digital control systems issues related to this work. It discusses several approaches to implementing digital controllers and reviews relevant past work. The objectives are:

- To review relevant background on digital control systems
- To assess current approaches used to implement digital controllers
- To review relevant past work on special-purpose architectures specially those applied to control systems

2.2 CONTROL SYSTEMS

A control system comprises subsystems and processes (or plants) assembled for the purpose of controlling the output of processes [Nise00]. It provides an output or response for a given input or stimulus. Control systems are found in a wide range of applications, from home appliances to the aerospace industry.

Control systems can be closed loop or open loop. Figure 2.1 shows a simplified block diagram of a typical closed loop or feedback control system. The plant is the process to be controlled, the input represents a desired response, and the output is the actual response. The feedback path feeds the plant output back to the input side of the system; thus the system can correct the output to compensate the effects of any disturbance. Open loop systems do not include the feedback path, making them unable to monitor or compensate any disturbances, although they are simpler and less expensive.

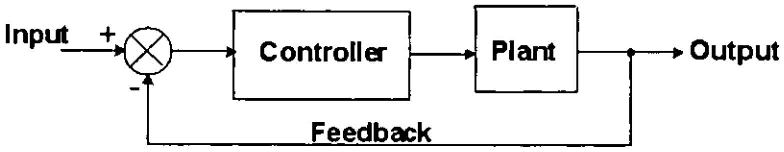


Figure 2.1 Block diagram of a typical feedback control system

2.2.1 Approaches to implement control systems

The controller shown in Figure 2.1 is usually an electronic circuit that operates on an analogue signal and outputs the same type of signal. In this case, the system is known as an analogue control system. Analogue systems operate in real time and are capable of a very high bandwidth, which is equivalent to having an infinite sampling frequency, so that the controller is effective at all times. However, their elements are usually hard-wired, so that their characteristics are fixed, making it more difficult to make design changes. Component ageing and sensitivity to environmental changes can be quite severe. Analogue components are also susceptible to noise problems.

As Figure 2.2 indicates, digital controllers can replace analogue controllers in control system applications. A continuous input signal in sampled by an analogue-to-digital (A/D) converter to produce a sequence of pulses, which are then used as input to digital controller. Then, the outputs of the digital controller drive the plant after they are converted to analogue signals by the digital-to-analogue (D/A) converter.

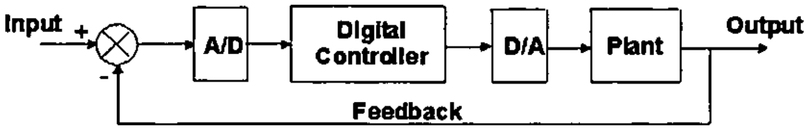


Figure 2.2 Block diagram of a typical digital feedback control system

2.2.2 Controller design

Despite the contrasts described above between digital and analogue control systems, the techniques used to design and analyse both type of systems exhibit some similarities. When modelling controller for a physical system, the designer converts the description of the system into a mathematical model, which then can be implemented in several ways.

Two approaches are available for the analysis and design of feedback control systems. The first is known as the classical, or frequency-domain, technique. This approach is based on converting a system's differential equation to a transfer function, thus generating a mathematical model that algebraically relates a representation of the output to a representation of the input. An advantage of these techniques is that they rapidly provide stability and transient response information. The primary disadvantage of the classical approach is its limited applicability: it can be applied only to linear, time-invariant systems or systems that can be approximated as such [Nise00].

The digital and analogue control systems are usually expressed in terms or frequency-domain transfer functions that are ratios of Laplace transforms or z-transforms [Middleton90]. These mathematical models describe the system's input-output relationships. Design and analysis of such systems using techniques such root-locus, Nyquist and Bode, is known as classical control theory, which has existed for more that 50 years [Nekoogar99, Nise00].

We can also represent digital and analogue control systems in the time domain by employing state-variable techniques and state-space models. These models also describe the input-output relationships, but additionally provide an internal description of the system. These models are especially useful when modelling multiple-input multiple-output (MIMO) systems. State-variable techniques and state-space models are included is modern control theory [Nise00].

2.2.3 State-space approach

The state-space approach is a unified method for modelling, analysing, and designing a wide range of systems [Nise00]. Although this representation of the system still involves a relationship between the input and output signals, it also involves an additional set of variables, called state variables. The mathematical equations describing the system, its input, and its outputs are usually divided in two parts:

- A set of mathematical equations relating the state variables to the input signal

- A second set of mathematical equations relating the state variables and the current input to the output signal

The essential matter when implementing a controller is performance and although they require more complex mathematics than transfer functions, state-space methods provide better performance than classical methods. The state variables provide information about all the internal signals in the system. As a result, the state-space description provides a more detailed description of the system than the input-output description. It can be used to represent non-linear systems. Also, it can handle, conveniently, systems with nonzero initial conditions. Time-variant systems can be represented in state-space. Many systems do not just have a single input and a single output. Multiple-input, multiple output systems can be compactly represented in state space with a model similar in form and complexity to that used for single-input, single-output systems.

2.3 DESIRED CHARACTERISTIC OF A DIGITAL CONTROLLER

Normally, control algorithms are well defined and present many characteristics that can be exploited to achieve efficient execution. Some of the desired features of the digital controller are explained below.

2.3.1 Fast multiply-accumulate operations

The most common arithmetic operation to be performed is of the type

$$y = \sum_{i=1}^{N} a_i x_i \tag{2.1}$$

Here, a sequence of N signal values a_i is multiplied by a corresponding sequence of signal values x_i, for i=1, 2, ...,N.

The number of input/output operation is relatively small when compared with the number of arithmetic operations. These features imply that the main load falls on the arithmetic unit (AU), and therefore it is essential to design an AU that executes operations on sum of products efficiently.

2.3.2 Memory bandwidth

An important consideration is the bandwidth of the data transfer between memory and the AU. It is crucial that the memory-AU bandwidth and the AU execution rate are balanced, otherwise one will be idle waiting for the other to finish its tasks. This means that the controller must complete several accesses to memory in a single instruction cycle, namely, fetching an instruction while simultaneously fetching operands for the instruction or storing the result of the previous instruction to memory.

2.3.3 Sample rate

A key characteristic of a digital control system is the sample rate. It is the rate at which analogue input values at sampled or processed and combined with the algorithm complexity determines the required speed of the controller implementation.

From signal processing theory [Proakis96], we know that the minimum sampling frequency has to be greater than twice the highest frequency component of the signal. However the filter required to perform the reconstruction is infinite dimensional and also, strictly, real signals do not have bandwidth limits (that is, there are still small frequency components outside the bandwidth) [Middleton90]. Thus, whilst there are some similarities between sampling for signal processing applications and control systems, when implementing a digital control systems it is often required to sample at a higher rate than the theoretical minimum [Feuer96] (at least 10 times). This has a direct effect in the processing power required by the digital controller.

In the previous paragraph we discussed that slow sampling usually results in poorer control performance. On the other extreme, excessively fast sampling results in similar loss in performance due to the difficulty to represent the small signal values involved in the calculations [Middleton90]. This is further explained in Sections 2.3.6 and 2.3.7.

The sampling rate depends on many signal-processing and system performance factors. The minimum sampling period is sometimes limited by the conversion times of the analogue-to-digital converters. It is important that the sampler device samples at a sufficiently fast rate so that the information contained in the input signal is not lost during the conversion. However, in complex control systems, the sampling rates may also be limited by the characteristics of the digital controller that processes the data, especially when modest devices are used.

The stability of a closed-loop digital control system is closely related to the sampling rate. The failure to maintain the necessary processing rates may result in a serious malfunction, this is because low sampling rates have a negative effect on the stability and as a consequence on the overall system performance. In such cases, the behaviour and output of the system would be impossible to predict. Therefore, the necessity of providing high sampling rate capability in control systems becomes evident.

2.3.4 Calculation delay

Independently of the technology selected to implement a digital controller, the computing time is nonzero. This situation is not important in many applications where real-time processing of data is not essential. However, real-time computation is necessary in control system applications, thus the time delays in handling the data and calculating the output response may have a significant effect on the system performance

Two immediate problems may be identified. Firstly, if the time delay is too large, there would not be enough time to complete the necessary computation required to complete the algorithm cycle before the next input sample is produced, and secondly, the time delay has an adverse effect on the stability of closed-loop control systems. Thus, the time delays can not always be neglected when implementing a digital controller.

In the case of programmable processors, the time delay may be obtained by analysing the program used to implement the control algorithm along with the subroutines that may be called. The number of instructions and the number of machine cycles required to execute them will determine the time delay for a particular algorithm. In general, it is possible to go through any program and with the information provided by the processor documentation, estimate the time necessary to complete the program or the time required to reach a particular point in the algorithm cycle.

2.3.5 Predictable, repeatable behaviour

To perform real-time control, the digital controller must have a predictable execution time. Also, it has to complete all the calculations and operations required for processing each sample before the next sample arrives. For instance, consider a system where the input is received at 20,000 samples per second, the controller must be able to maintain a sustained throughput of 20,000 samples per second. However, it is important not to make it faster than required. As speed increases, so does the cost, power consumption and design difficulty.

2.3.6 Fixed-point and floating-point arithmetic

Another important characteristic to determine the suitability of a digital controller for a given application is the type of binary numeric representation used by the processor. The numeric representation and the type of arithmetic used can have a profound influence on the behaviour and performance of the controller. One of the most important decisions taken by the control engineer when implementing an algorithm is between the use of fixed-point or floating-point arithmetic. Most microcontrollers and early-generation DSPs use fixed-point arithmetic in which only a finite amount of word length is available to represent the magnitude of the signal or coefficients. Thus, signals and coefficients must be scaled to fit the word length provided by the processor. Most of the devices currently used to implement digital control use fixed-point arithmetic, especially in cost-sensitive applications [Bdti00, Goodall00, Schlett98].

Fixed-point arithmetic represents the number in a fixed range with a finite number of bits of precision (word width). Any number outside of the specified range can not be represented. Floating-point arithmetic expands the available range of values. It represents the number in two parts: a mantissa and an exponent. The mantissa value lies between -1.0 and +1.0, while the exponent scales (in terms of powers of two) the mantissa value in order to create the actual value represented.

$$\text{value} = \text{mantissa} \times 2^{\text{exponent}}$$

One problem is when the width of the processor registers is not sufficient to hold the result of the filter arithmetic. This is similar to when the desired output of an analogue filter becomes larger than the supply. Thus, the choice of fixed-point or floating-point arithmetic is determined by the system requirements in terms of dynamic range and precision. The dynamic range is the ratio, usually expressed in dB, between the largest and smallest numbers that can be represented. Precision in a digital system is dependent upon the accuracy of the arithmetic used.

Floating-point arithmetic offers an ease-of-use advantage due to the fact that in many cases dynamic range and precision are not concern. This increase of dynamic range also allows a designer to ignore scaling problems because it reduces the probability of overflow. In contrast, on fixed-point processors, sometimes it is necessary to scale signals at various stages of the program to ensure adequate numeric performance. Unfortunately, floating-point arithmetic is generally slower, more expensive and more difficult to implement in hardware. The increased cost results from the more complex circuitry required. In addition, the larger word sizes of floating-point processors often means that memory and buses are wider, raising the overall system cost.

2.3.7 Effects of finite word length and quantisation

In general, when implementing a digital controller using a general-purpose processor, the value of the input signals is quantised and the internal state variables are truncated when their next value is calculated. This is because of the finite word lengths used to represent the magnitude of the signals. Thus, it is necessary to scale the values of the signals, coefficients and state variables to fit the word length of the processor. This can have a profound influence on the behaviour and performance of the control system.

To implement a digital controller, it is necessary to map the control algorithm into some kind of architecture that will actually perform the task. There are many alternatives, it might be implemented in software on general-purpose processors, microcontrollers, or digital signal processors, or it might be implemented in special-purpose processors. Control applications may also take advantage of entire platforms built around general-purpose processors. Personal computers, workstations or stand-alone boards are among these platforms.

2.4.1 General purpose processors

In principle, all digital control algorithms can be implemented by programming general-purpose processor, but this solution is not cost effective in many applications, and often the performance requirements in terms of throughput, power consumption, and size cannot be met [Bdti00, Irwin98]. The reason for this is the mismatch between general-purpose processor architectures and most control algorithms that require a large number of repeated arithmetic operations of a relatively simple nature and a low number of input/output operations.

General-purpose processors are designed to perform a multitude of functions to support applications such as word processing and similar programs that rely almost entirely on manipulation of data; this involves storing, organising, sorting and retrieving information. To perform those tasks, the processors provide a number of functions that allow wide-ranging mixtures of operations and control flow that can be data dependent, making large jumps from one area of the program memory to another. Thus, the ability to move data from one location to another and testing for inequalities (A=B, A<B, etc.) becomes essential [Lapsley97].

These processors were not originally designed for multiplication-intensive tasks, even some modern processors would require several instruction cycles to complete a multiplication because they do not have dedicated hardware for single-cycle multiplication, as a consequence they are not well suited to perform control algorithms [Bdti00]. To solve this problem, high-end processors such as Pentiums and PowerPCs, have been enhanced to increase the computation of arithmetic-intensive tasks. A common modification is the addition of SIMD-based instruction set extensions that take advantage of wide resources such as buses, registers and ALUs, which can be seen as multiple smaller resources. For example, a 64-bit data bus can handle 4 different 16-bit words simultaneously. However, despite the high performance operation offered by these processors, they are not widely used in embedded applications due mainly to their cost [Eyre00].

2.4.2 Digital signal processors

Digital signal processors (DSP) have been designed to overcome some of the limitations found general-purpose processors. DSPs introduce some architectural features that accelerate the execution of repetitive multiply-accumulate operations of digital control algorithms [Eyre98].

Among the main DSP features are:

- Hardware multipliers that can handle the multiplication and accumulate operations rapidly, generally in one instruction cycle. An instruction cycle is usually one or two clock cycles long (RISC-like architecture).

- Several functional units that perform some sort of parallel processing.

- Harvard bus architecture that provides high memory bandwidth to allow simultaneous processing of program instructions and data.

- Internal memory organisation, usually involving more than one large on-chip memories that can be accessed once every instruction cycle and are used to store data, instructions, or look-up tables.

- Specialised addressing modes such as circular addressing and pre- and post-modification of address pointers.

- Large number of internal registers.

- Typically, a multiplication and addition, data fetch, instructions fetch and decode, and memory pointer increment can be done simultaneously.

The high-speed capability of the DSP allows the device to be applied to adaptive control, in which case, the processor can simultaneously perform monitoring and control functions. DSPs can be used for controlling external digital hardware as well as processing the input signals and formulating appropriate output signals. Although most real-time digital control applications require large amount of data calculations, the programs that implement them are normally very simple. As a result, these programs can be stored in internal memory to reduce the transfer time. The design process involves mainly coding the control algorithm either using a high-level language or directly in assembly language. Then, the source code is compiled into an object code that can be executed by the processor.

This approach allows rapid prototyping, but unfortunately it is not always possible to meet the requirements of power consumption, size, or cost. The main reason is that the standard DSP is designed to be flexible in order to support a wide range of digital signal processing algorithms while most algorithms use only a few of the instructions provided [Lapsley97].

2.4.3 Microcontrollers

Unlike general-purpose processors that are designed to support large word width and address spaces, a microcontroller design is focused on integrating the peripherals needed to provide control within an embedded environment. Commonly, a microcontroller incorporates in a single chip at least the necessary components of a complete computer system: CPU, memory, clock oscillator and input and output ports, plus some additional elements such as timers, serial units, and analogue-digital and digital-analogue converters. These features allow them to be simply wired into a circuit with very little support requirements; usually, they only require power and clocking. Maximum speeds for the different devices are typically in the lows tens of megahertz [Predko99, Cady97].

The primary role of microcontrollers is to provide inexpensive, programmable logic control and interfacing to external devices. Thus, they are not expected to provide arithmetic-intensive functions. When included within complex systems applications, they are used to interpret input (from a user or from the environment), communicate with other devices, and output data to a variety of different devices. Microcontrollers add a great deal of flexibility in the product development process as they can be used for a variety of applications. Another advantage is the fact that microcontrollers are member of families that present many different combinations of hardware features, so the most suitable device for a specific application can be selected. Some, especially those with 16- or 32-bit data paths, rely almost completely on external memory. The external memory contains program to be executed, usually in ROM, as well as the RAM required for the application.

2.4.4 General purpose parallel processors

Some multiprocessing approaches have been proposed to satisfy the demands of very complex control systems. These alternative strategies can be based in multi-processor or multi-computer systems. The difference between these two categories lies in the way in which communication between the processors is organised. All the processing elements share the same memory in a multiprocessor system while in a multi-computer system, each processor has its own private memory space [Wanhammar99].

The main challenge of this approach is to distribute the computational load across the processing elements so the execution time is reduced to a minimum. Thus, it is necessary to match the computational requirements of the algorithm with the available hardware resources and minimise the communication among processing elements. Although any processor can potentially be used in a parallel processor design, it is desired that the selected processors include some additional features such as multiple external buses, bus-sharing logic, and multiple parallel dedicated ports designed to simplify the interprocessor communication so the overall system performance is not affected [Lapsley97].

2.4.5 Fuzzy logic controllers

Another approach to implement digital control involves the use of fuzzy logic controllers (FLC), which can be used together with both state-variable and classical techniques [Patyra96]. Fuzzy logic controllers can be applied to systems with undefined boundaries that are difficult to represent using explicit difference or differential equation descriptions. Most applications of fuzzy logic have low computational loads, so hardware designs implement fuzzy logic using general-purpose controllers. However, as new applications emerge, traditional approaches may not cover all the systems needs and some dedicated architectures that specialise in fuzzy computation have been proposed. These new architectures support fuzzy logic applications efficiently, but their main drawback is the difficulty to adapt them to different applications [Costa97]. This is due to their fixed features, such as the number of input and outputs variables, the value resolution, and other fuzzy control parameters.

2.4.6 Special purpose processors

All the approaches to implement digital controllers discussed above use existing architectures to map the control algorithm, via programming, to fit the architecture. But it is also possible to change the architecture to better suit the algorithm. Special purpose processors, with a particular combination of registers, logic elements and interconnections, open the possibility of achieving in one clock cycle what a traditional programmable processors require tens or even hundreds of clock cycles.

The term special-purpose processor has been used to define a wide range of degrees of dedication and specialisation. We can say that a special-purpose digital control processor is a dedicated hardware entity whose function is to perform a specific, well defined, set of digital control algorithms in real-time. Just as DSPs are more efficient and cost-effective than general-purpose processors to execute high-speed arithmetic operations, special-purpose processors have the potential of overpower DSPs due to its specialised nature. As only the required functions are placed in hardware, special-purpose processors can be less expensive than other processors, especially for high-volume products.

There is not a single correct solution to the problem of designing a processing system that meets the needs of real-time control. Instead, the resulting system is defined by a series of trade-off decisions taken by the designer when mapping the algorithm to the final solution within the constraints imposed by the system requirements.

The possibility of integrating a whole control system into one chip has several effects. It increases the processing capacity and simultaneously reduces the size of the system, power consumption, and pin restriction problems. Additionally, it improves system reliability and offers protection of intellectual property. Of course, developing special-purpose architectures presents some drawbacks. Among them are the effort and expense associated with custom hardware development, especially for custom chip design. However, the problems associated with custom hardware can be partially solved using high-level hardware design languages such as VHDL and logic synthesis CAD suites allied to large low-cost reprogrammable FPGAs.

A major advantage of this approach is that the data word length can be adjusted to the systems requirements. Thus, the size of the architecture can be kept to a minimum. However, the performance improvements come with the cost of larger design effort.

2.4.7 Combined approaches

Digital signal processors are paired with microcontrollers in many applications. As some systems that utilise digital system processing in their operation also have digital control processing requirements, there are some devices that combine the features provided by both processors into a single solution.

This approach looks into the integration of the DSP functionality with the microcontroller to offer the benefits of the two architectures. Using a single processor to implement both types of software is attractive, because it can potentially simplify the design task, save board space, reduce total power consumption and reduce overall system cost. In order to use these new devices, the system designer must evaluate what performance is needed to control the system and what performance is needed to perform the signal processing [Eyre00].

2.5 SOFTWARE ISSUES

This section describes software issues related to the implementation of digital control systems.

2.5.1 Software structure

Programs that implement control algorithms are different from traditional software applications in two main aspects. First, the programs are usually shorter, normally counted in tens or hundreds of lines versus tens of thousands lines [Lapsley97]. Second, the execution speed is often a critical part of the application. Typically, the overall structure of the software consists of a main program that performs an initialisation process and then executes one or more control loops that perform the operations defined by the control algorithm.

2.5.2 Programming languages

The traditional language to write programs that implement control algorithms is C, mainly because the programs are easier to develop and maintain that those programmed using assembly language. Another key advantage is that the programmer does need to understand the architecture of the processor being used. When execution speed is important, some critical programs or subroutines are programmed using assembly code, however, this requires that the programmer have deeper knowledge of the architecture. Thus, the choice of using assembly or C depends largely on what is more important for the application, performance or flexibility and fast development. Existing software modules can be reused to minimise the cost of developing new applications. This approach is effective when a library of optimised modules has been accumulated form past designs so new applications can be constructed with segments of existing modules. Other factors to be considered are the complexity of the control algorithm, compiler efficiency, team experience, and manpower.

The programming of a processor usually requires knowledge of the specific processor assembler language. The support for a high-level computer language is usually via the compilation of the high-level language program into the target assembler. However, the efficiency of those programs is highly dependent on the compiler technology. Therefore, for the sake of exploiting the fullest possible processing power and memory efficiency, some processors programs are handcrafted with little emphasis on the programming structure. The design and debugging process may take months to complete and because the programming skills tend to be very specialised and take time to acquire, the resulting handcrafted programs tend to be not only device-specific but also programmer-specific. As a result, these programs are hard to maintain and even harder to modify. Therefore, it is essential to devise a design route that would allow algorithmic ideas to be implemented efficiently.

2.5.3 Numerical subroutines

Timing analysis of digital control algorithms programmed on a general-purpose processor may reveal bottlenecks, or small portions of code that contribute disproportionately to execution time. These bottlenecks may be repeated many times as the program progresses from start to finish. Figure 2.3 shows a typical execution profile [Ackenhusen99], or plot of instruction address versus time, of a program that performs a control algorithm. The algorithm begins at its initial instruction (a), normally an initialisation process that progress lineally for a small amount of instructions, then jumps to a subroutine at a higher address value where it executes a specific task (b). The program then exits the subroutine (c), progress a bit further within the program (d) and then it jumps again the subroutine and so on until the program completes (e) or starts a new execution loop (f). In addition, associated with each subroutine call is some time-consuming overhead. Data and control register values must be stored, usually in a stack, and a pointer must be set so that when the subroutine execution is completed, the main program may resume its execution at the point it was left before the subroutine was called. Also, input parameters and output results must be passed from/to the subroutine.

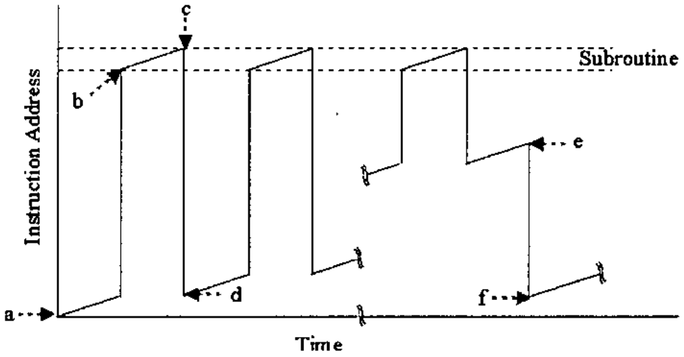


Figure 2.3 Typical plot of location of instruction being executed versus time of a program that performs a control algorithm

2.6 LITERATURE SURVEY

In the past there has been much work on architectures for control applications. For the purpose of this revision, the architectures are divided into the following categories: general-purpose, specialised and reconfigurable architectures.

2.6.1 General-purpose architectures

General-purpose processors are designed to satisfy the requirements of control systems in general. The algorithms to be implemented using these devices are mapped, via programming, to fit the architecture. [Lang84] describes the design of a special-purpose digital processor targeted for control system implementations. It uses logarithmic arithmetic to improve the computational dynamic range, accuracy and speed. [Jaswa85] describes a reduced instruction set coprocessor that optimises the states transitions of the controller. It consists of continuous processing elements capable of performing the next-state update process and a discrete processing element for processing switch-based information.

[Agrawal95] presents a system design of an industrial controller that can be customised for specific tasks. The design is based on a commercially available controller that can be customised for any specific needs. The customised functionality is achieved in software and then ported into the controller. [Nadehara95] describes a 32-bit RISC microprocessor designed for software signal processing. The instruction set is oriented towards signal processing, it includes fast integer/fixed-point multiply/multiply-accumulate instructions. The processor integrates a 32-bit compact multiply-adder with a parallel overflow detector in its pipeline to achieve peak signal processing performance. Some designs are based on existing processor cores; [Furber99] describes an asynchronous controller for small embedded systems. The system chip incorporates a 32-bit asynchronous RISC processor core, a 4-Kb pipelined cache, a flexible memory interface, and assorted programmable control functions.

Some parallel designs have been proposed to control demanding complex processes. [Tokhi95] presents an investigation into the utilisation of parallel digital signal processing devices for real-time control. It discusses the issues of algorithm parallelisation and hardware mapping. [Darbyshire95] describes the features of a DSP system designed for large-scale active control. The DSP system presented has a multiprocessor architecture integrated as a real-time processor with multiple analogue input and output channels. It incorporates commercial DSPs as basic processing elements.

Other designs are based on fuzzy logic techniques; [Patyra96] discusses various aspects of digital fuzzy logic controller (FLC) design and implementation. It analyses classic and improved models of the single-input single-output (SISO), multiple-input single-output (MISO), and multiple-input multiple-output (MIMO) in terms of hardware cost and performance. It also illustrates the improved implementation of highly parallel FLC in digital technique. [Costa97] proposes an architecture dedicated mainly to medium-range applications that demand computational power combined with low cost for the resulting hardware system. The architecture is a 16-bit processor with dedicated instructions and hardware for support of fuzzy logic.

2.6.2 Dedicated architectures

Dedicated architectures are designed to solve one specific task. In [Ling88] a VLSI robotics vector processor for real-time control is described. The processor has three floating-point processors, each with an adder, multiplier and register file, all operating in a SIMD fashion. It employs a RISC-like architecture with seven basic instructions. [Liu91] proposes an integrated solution to compute real-time robot control using a special-purpose VLSI array. The array is connected in a systolic manner using different types of basic processing elements. Also applied to robot control, Catthoor[91] describes the design of an application-specific architecture that consists of four execution units and a data RAM. The application is a six-degree-of-freedom mechanical robot for industrial applications.

[Garberg96] considers the use of an ASIC for a stand-alone controller. A control system that controls an inverted pendulum is designed using software tools and then mapped into hardware. The resulting controller estimates the process-states and controls the dynamic process. The same authors present a similar controller implemented in an FPGA [Garberg98]. [Samet98] presents a comparative study where a PID algorithm is mapped into three different hardware architectures, which perform the arithmetic operations for the PID controller in serial, parallel and mixed form.

[Grout95] Describes an ASIC which the functionality of a digital proportional plus integral (PI) error actuated controller with auxiliary feedback. The controller provides discrete time control of a range of continuous time systems by receiving analogue inputs via a single multiplexed analogue to digital converter and providing an analogue output via a digital to analogue converter. It allows both proportional and integral gains to be adjustable using a digital control word.

2.6.3 Reconfigurable architectures

Reconfigurable architectures are those that can be adapted to satisfy the requirements of control algorithms. A number of these architectures have been proposed. [Fujioka96] proposes a reconfigurable parallel processor to reduce the delay time of multi-operand multiply-additions performed in the sensor feedback control of intelligent robots. In each PE, a switch circuit is used to change the connection between multipliers and adders. The multiply-adders can be reconfigured every clock cycle using a very-long-instruction-word (VLIW) control method. [Tsunekawa95] proposes a VLSI-oriented highly parallel architecture for state-space digital filters, where multiple processing elements (PE) are combined to implement the state-space equations. [Chen91] describes the design of Programmable Arithmetic Devices for DIgital signal processing (PADDI). It is a programmable medium-grained device that supports the implementation of algorithmic specific data paths for real-time signal processing applications. It contains 32 16-bit execution units (EXU), each with its own instruction nano-store. The execution units are connected by a configurable hierarchical switch, which enables both pipelined and parallel operation. A similar device, programmable adaptive computing engine (PACE) is proposed in [Spray91]. It is a medium-grained cellular automation-based architecture that supports regularly and irregularly structured functions within a regularly structured array. Example implementations of three irregularly structured algorithms including a PID controller are explained.

Another fully reconfigurable approach is described in [Herpel93]; it presents a custom computer together with a software environment for implementation of algorithms for real-time control. The custom computer is based on FPGA boards embedded in programmable interconnection network. The transformation of an algorithmic system specification into a configuration file for the FPGAs is done through a set of high-level and structural synthesis tools. The software tools allow prototype implementation of algorithms on the reconfigurable hardware that is used to validate the design before an ASIC implementation.

Fuzzy logic offers the possibility of dynamic configuration. In [Dettlof89] a general-purpose fuzzy logic inference engine for real-time control applications is presented. A TTL compatible host interface downloads the rules into the fuzzy memory at boot-time, and can also update the rules dynamically to reconfigure the controller. A similar approach is used in [Donald94]; it describes a custom designed hardware fuzzy logic controller (FLC) for high-speed real-time control applications. It has a pipelined architecture and its knowledge base can be updated at run time by a supervisory microprocessor that constantly monitors the FLC's performance and update the FLC's knowledge base at run time when dealing with changing environment and plant characteristics. In both of the previous controllers, the functionality of the controller can be adapted to changes, although the hardware structure remains unaltered.

Moving towards neural network approaches, [Liu99] presents a parallel learning neural network chip, which is used to perform real-time output feedback control of a nonlinear dynamic plant. The proposed hardware utilises parallelism to achieve speed independent of the size of the network, enabling real-time control. The on-chip learning ability allows the hardware neural network to learn on-line as the plant is running and the plant parameters are changing. This adaptive controller does not need any prior knowledge of the system. [Palmer94] presents an architecture and development environment for a family of neural network processors targeted at real-data formats, word length requirements, and sample frequency is needed. The aim of this study is to design a custom architecture, which can meet the requirements of the control systems to be implemented.

By adapting the processor architecture to the requirements of the control algorithm, we aim to achieve in 1 clock cycle what traditional programmable architectures requires tens, or more, clock cycles to complete (see Section 7.5.2). Also, taking into account the system requirements when designing the architecture should ensure that the architecture performs deterministically in a real-time embedded control environment. Cost savings also motivate the use of a custom processor, as the resulting architecture is likely to be smaller when compared with general-purpose processors. The design of a custom architecture includes additional design decisions when compared with a dedicated architecture. One of the main issues involves the definition of an instruction set.

2.7 SUMMARY AND CONCLUSIONS OF THE CHAPTER

This chapter has introduced the current approaches to implement digital control systems, identifying their advantages and drawbacks. It also highlighted the potential benefits of using special-purpose architectures to implement such systems. Finally, related work on hardware architectures, especially those applied to control systems, has been reviewed.

The rising popularity of signal processing applications has led designers to add signal processing capabilities to existing processors in order to support computation-intensive tasks. Thus, while the points explained in Section 2.4 traditionally distinguish general-purpose processors, DSPs and microcontrollers, it is important to realise that the line that divides these devices is fading. It is now common to find general-purpose processors that include DSP features or DSPs with microcontroller's capabilities.

Digital control systems have physical limitations due to the nature of the system components. For example, the sampling period is determined by the clock frequency and how fast the numerical operations and instructions are executed by the processor. Another important issue is that all numbers can be represented only with finite precision.

The main advantage of using of general-purpose processors to implement control systems is that the algorithm can be modified relatively easy if required by changing the software program. General-purpose processor architectures often require several instructions to perform operations that can be performed with just one DSP processor instruction, but run at faster speeds. In general, general-purpose processors are a good option when implementing applications that require both, DSP and non-DSP processing. Furthermore, the most popular general-purpose processors are supported by a large variety of application development tools. However, when general-purpose processors are used only for computation-intensive tasks, they are rarely cost-effective compared with DSP processors [Bdti00].

In its initial form, the control algorithms take full advantage of any inherent parallelism and have not regard to any potential implementation consideration. It is only when the algorithm has to be mapped into hardware that some trade-offs have to be made. When implementing control systems using standard programmable processors, the control algorithm has to be artificially partitioned and constrained to meet the physical bus widths and mapped on the instruction set, the system performance may be affected.

Custom hardware normally offers the most efficient implementation because the hardware architecture is designed to match the algorithm [Wanhammar99]. It also offers the possibility of integrating many functions within a single device. However, it is generally time-consuming and expensive to develop, although the cost-per-unit is low when produced in volume.

From this chapter we can conclude that:

- There is not a 'best' approach to implement real-time high performance control systems. No one processor or custom architecture can meet the needs of most applications. Several factors like cost, performance, integration, easy of development, power consumption, development tools, will determine which option is the most suitable to implement a specific control system.

- If a standard processor can meet the requirements of a particular application, it is often the best approach. This allows a fast implementation and the system can be easily modified or new features can be added.

- Custom architectures offer potential cost-performance benefits when used to implement complex control systems that require very high sample rates.

Chapter 3

Research overview

3.1 OBJECTIVES OF THE CHAPTER

This chapter presents an overview of the investigations of this thesis. The objectives are:

- To identify the areas that need to be investigated and introduce the experimental work to be carried out
- To describe the ProASIC design flow and the design and verification tools used during the implementation of work
- To establish the design and experimental assumptions when implementing and verifying the CSP

3.2 IDENTIFICATION OF INVESTIGATIONS

This section identifies the investigations needed to fulfil the objectives of this research.

- Control algorithm: to identify a filter structure that performs the control algorithm, which may be exploited to reduce the number of required operations and is suitable for hardware implementation. This involves an analysis of the set and sequence of arithmetic operations, the data set, numerical accuracy of the coefficients, word length for the internal variables, and numeric formats.

- Architecture: a design strategy to implement the hardware architecture that will perform the control algorithm has to be selected. It is necessary to define a set of basic operations that can be executed on the processing elements, the data set to be stored in the storage elements, the interface and connections, and to define a control strategy to co-ordinate activities between the architectural components. The design area must be of a size that allows its implementation using the selected technology. A performance analysis of the system implementation may lead to proposed architecture modifications that reduce the hardware costs and/or processing time.

- Program structure: An overall structure of the program that implements the control algorithm has to be defined. This structure will determine the order in which processes such as initialisation, control loops and input sampling have to be implemented.

- System evaluation: this involves the identification of a comprehensive set of tests to prove the CSP's operation for a variety of filter types over a range of input conditions. The results of these tests will be used to benchmark the CSP performance against other processors.

3.3 METHODOLOGY

Figure 3.1 shows a diagram of the design methodology. Firstly, we identify a filter structure to perform the control algorithm, which may be exploited to reduce the number of required operations and is suitable for hardware implementation. This involves an analysis of the set and sequence of arithmetic operations, data set, numerical accuracy of the coefficients and internal variables, and numeric formats.

Once the filter characteristics and implementation requirements have been identified, a software model of the processor is created in Java; its purpose is to provide a a clear understanding of the algorithm and its numerical requirements, as well as a functional specification of the processor and test vectors to verify the hardware design. The CSP model architecture is modular; this modularity allows to replace processing elements to undertake performance comparisons and to explore new architectures. Furthermore, the model supports alternative algorithms, thus making it suitable for demonstration purposes in a range of control application environments. The results of the model running several control algorithms are compared against the results obtained from MATLAB programs that implement the same control algorithms using IEEE 32-bit floating-point format to represent the coefficients and state variables.

When the simulation results of the software model are correct, the next step is to create a hardware model of the CSP using the hardware description language VHDL. The hardware model is simulated and verified using the information generated by the Java model

The hardware model is simulated and verified using test vectors generated by the software model. Then, it is synthesised, and as a final verification before programming a device, the synthesised netlist is simulated and using the VHDL testbench and test vectors as before. Finally, the hardware implementation is verified using a hardware tester and the same test vectors.

3.4 EXPERIMENTAL VEHICLE DESCRIPTION

This section describes the environment in which the experiments are carried out and the assumptions made to perform the experiments.

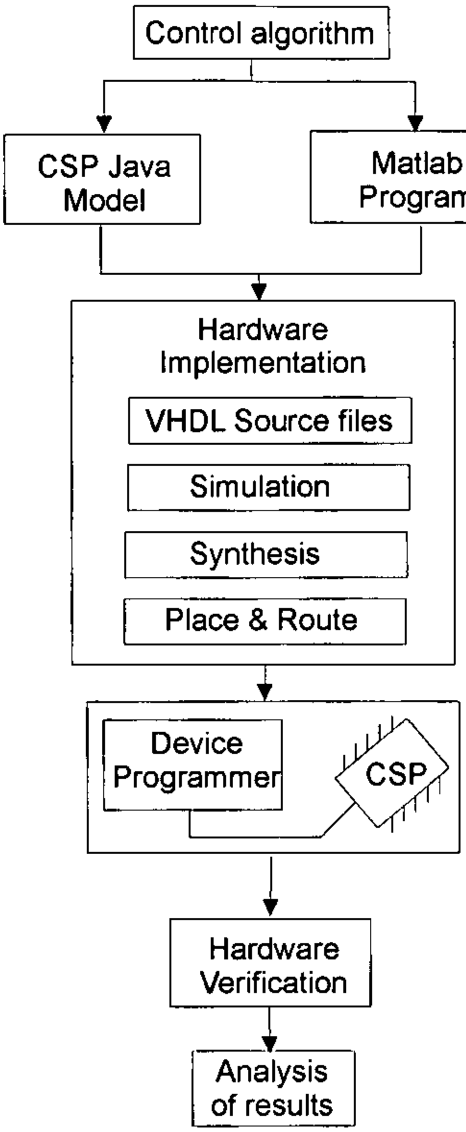


Figure 3.1 Summary of design methodology

3.4.1 Software simulation

The software model was programmed and simulated in the high-level language Java using the programming environment provided by Microsoft Developer Studio. The advantages of using Java for software simulations are:

- The language is robust and versatile
- The source code is platform independent
- It is easy to program and debug

3.4.2 Technology

A ProASIC A500K130 programmable device from Actel has been used to implement the hardware design [Actel00a]. The ProASIC device core consists of a Sea-of-Tiles (Figure 3.2). The basic logic unit consists of a programmable three input, one output cell or tile. Each logic tile can be configured into a 3-input logic function (e.g. NAND gate, D-Flip-Flop, etc.). The A500K130 has a total of 12,800 tiles.

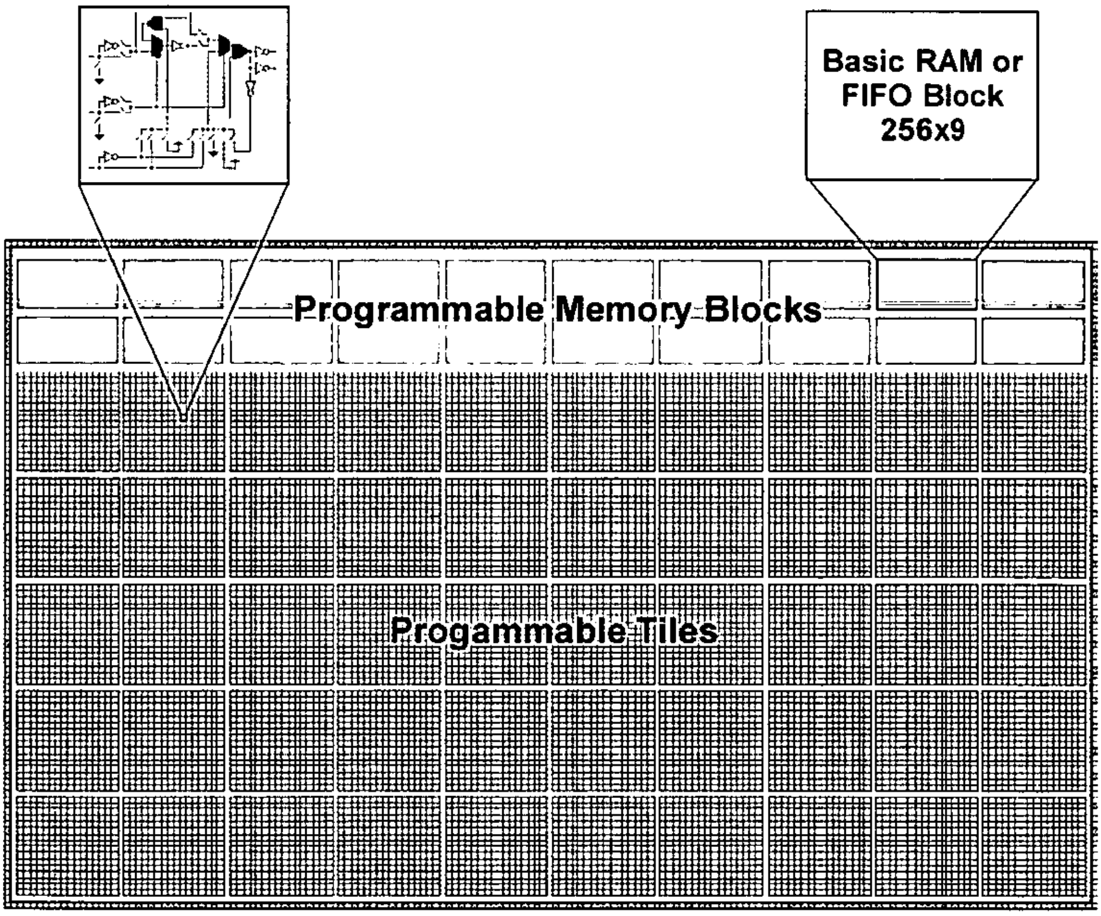


Figure 3.2 Actel ProASIC architecture

ProASIC devices provide two alternatives to implement memories: embedded and distributed memories [Actel00b]. Embedded memories use, as their name indicates, dedicated embedded memory blocks; while distributed memories are implemented using core logic tiles.

The devices contain embedded two-port SRAM memory blocks that have built in FIFO/RAM control logic. The memory blocks are located across the top of the device and depending upon the device, 6 to 28 blocks of memory are available. The A500K130 include 20 memory blocks (Figure 3.2). Each block can be configured independently and is 256 words deep and 9 bits wide. They have separated and independent read and write ports allowing simultaneous ports accesses. Embedded memories can be combined in parallel to form wider memories or stacked to form deeper memories.

Embedded memories can also be used to form multiple-access memories. Figure 3.3 shows an example of a 256-word x 9-bit multiported memory with two read and one write ports. For this example 2 memory blocks are required, with each block providing a read port. When a word is written into the memory, the incoming data is stored in both memory blocks. This means that each block will contain exactly the same data than the other, so that both read ports can access any word. Thus, the price paid for a multiport memory is a reduction in storage capability. Note that ProASIC devices do not support memory blocks with multiple writes.

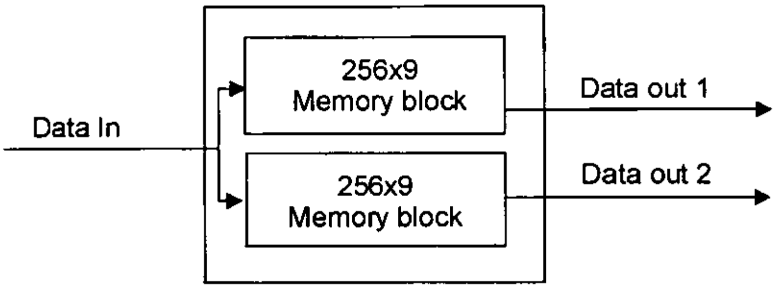


Figure 3.3 Example of a 256x9 two read one write memory

Distributed memories have independent asynchronous read and write ports, and are generally slower and larger compared to embedded memories. The maximum size of a distributed memory that can be implemented in a A500K130 device is 64 words, and each word comprising up to 78 bits. The manufacturer recommends that larger memories should be implemented using embedded memories.

3.4.3 Hardware design and simulation

The hardware model was programmed in VHDL using Computer Aided Engineering (CAE) tools from Veribest. With the purpose of verifying the functional correctness of the VHDL model, it is simulated together with a VHDL testbench which uses test vectors generated by the Java model to drive the inputs of the hardware model and then compares the actual outputs against the expected outputs (Figure 3.4). The testbench will indicate if the simulation was successful and in case of failure, will help to identify possible errors in code the in order to correct the design.

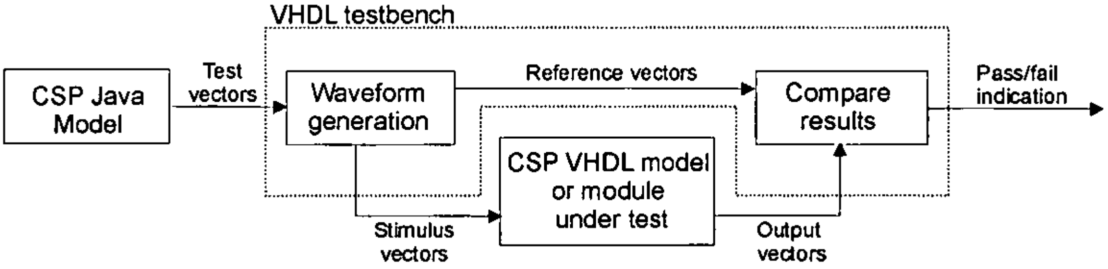


Figure 3.4 Testbench structure

Once the results of the VHDL simulation are correct, this verified hardware model is synthesised using the Leonardo Sprectrum synthesiser from Exemplar Logic. The synthesiser produces a technology specific netlist in VHDL format. This VHDL netlist can be used to perform post-synthesis simulation to perform further verification of the design. The VHDL netlist is then used as input for the Actel's ASICmaster that performs timing-driven placed and route. The ASICmaster tool includes a power estimator and provides back annotated delay information for post-place and route simulation and static timing analysis. A performance analysis of the hardware design is done using the static analyser Flash Timer from Actel. Figure 3.5 summarises the Actel's ProASIC design flow.

3.4.4 Hardware verification

The ASICmaster also produces a bitstream file that is used by the Silicon Sculptor to program a ProASIC device. The Silicon Sculptor is a single device programmer with stand alone software for the PC.

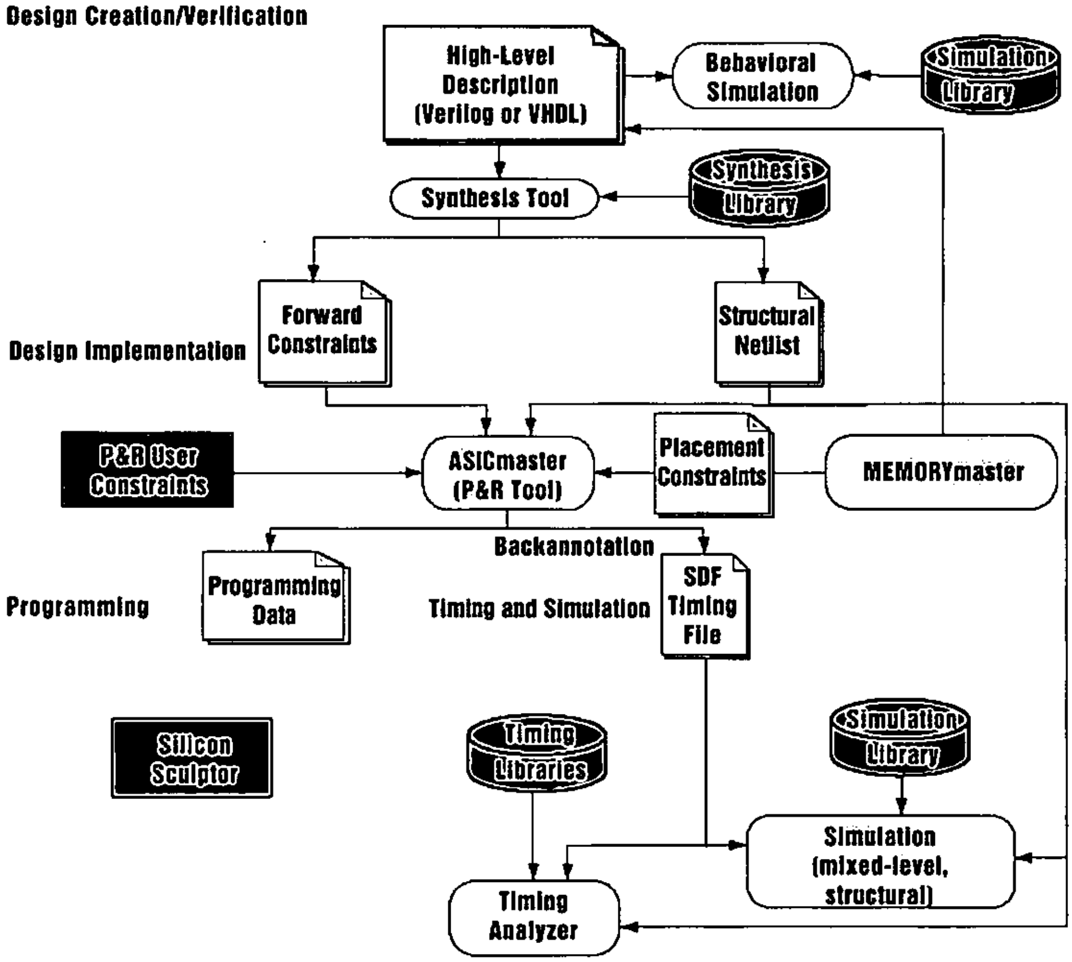


Figure 3.5 ProASIC design flow [Actel00a]

When the device has been successfully programmed, a serial tester is used to verify the behaviour of the design on the ProASIC device (Figure 3.6). The serial tester uses the build-in JTAG circuitry of the ProASIC device to place the input signals to the input pins of the device and to read the output signals form the output pins. A set of inputs is used to drive the device on every clock cycle. The outputs are then compared against the expected outputs. The test vectors used in this process are the same vectors used to for the hardware simulation and were generated by the Java model.

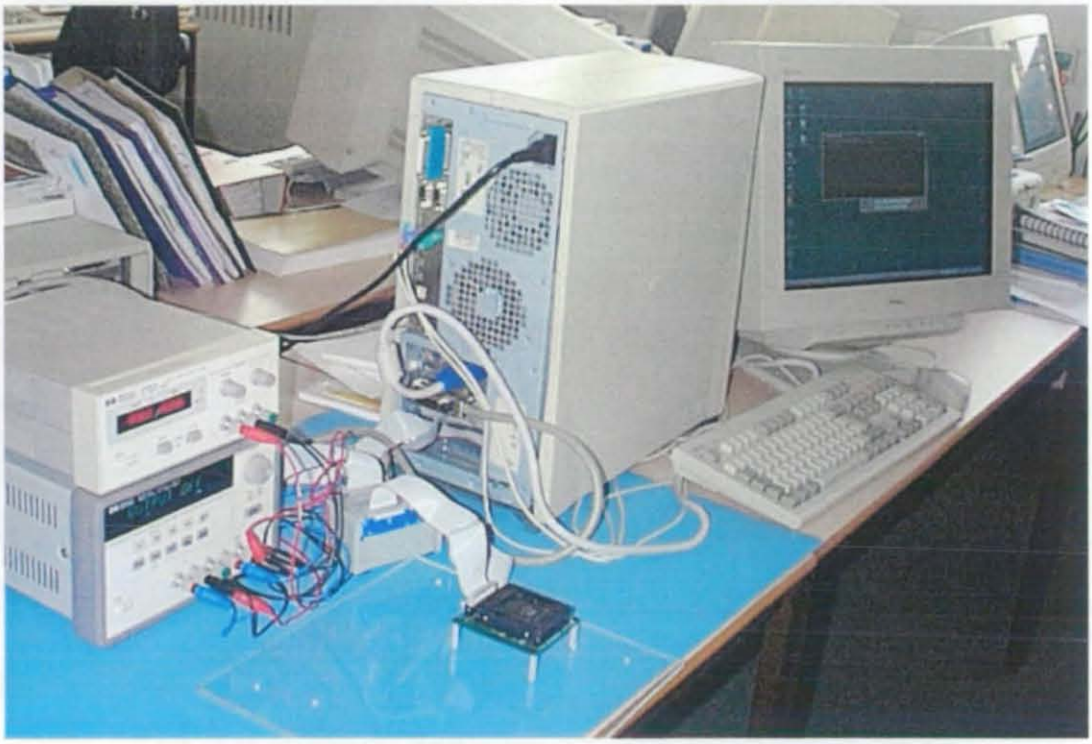


Figure 3.6 Serial tester

3.5 Design and experimental assumptions

This section outlines the assumptions taken when design the CSP and performing the experiments.

- The number of bits per sample is determined by the of the analogue-digital and digital-analogue conversion hardware used. Between 8 and 12 bits are common for control applications, thus a word length of 12 bits has been adopted such that the controller can be used for general application.

- As the processor is targeted towards linear time-invariant control, it is assumed that the controller's behaviour does not change over time.

- Input and output values are synchronised with the same sample period. This means that a set of outputs will generated for each set of inputs.

- The deadline to perform the operations is determined by the sampling period.

Chapter 4

Controller formulation

4.1 Objectives of the chapter

This chapter introduces the controller formulation selected to implement the control algorithm. The objectives are:

- To identify digital filter structures that minimise the number and complexity of operations needed to calculate the output values
- To define a suitable format to represent the coefficients and state variables
- To identify the properties of the selected filter structure that facilitate its hardware implementation

4.2 State-space description of control systems

For this research we use the modern approach [Nise00], which utilises the state-space formulation to represent a control system. Traditional approaches such as transfer functions, block diagrams, or signals flow graphs, involve a relationship between the input and output signals. Although the state-space representation of the system still involves such relationships, it also involves an additional set of variables, called state variables. The state variables provide information about the internal signals in the system. As a result, the state-space description provides a more convenient and powerful way of describing and dealing with systems than the input-output description.

The state-space formulation is more commonly used to represent the whole system, but for this work we are using it just to represent the controller that is to be implemented. Both continuous-time and discrete-time formulations are possible, but of course it is the discrete time version which is of interest here.

The mathematical equations describing the system, its inputs, and its outputs are usually divided in two parts:

1. A set of mathematical equations relating the state variables to the input signal (the 'state equation').

2. A second set of mathematical equations relating the state variables and the current input to the output signal (the 'output equation').

The state and output signals of a discrete system are found from the inputs and initial state. The state-space description offers a number of advantages [Nise00, Santina94] when compared to traditional approaches such as transfer functions, block diagrams, or signals flow graphs, including:

- It is a standard representation with simple notation

- It is an easy way of expressing equations for complex controllers

- Matrix algebra can be applied directly

- It allows a unified representation of multi-input and multi-output system models with similar form and complexity to that used for single-input, single-output systems

- It can readily handle systems with nonzero initial conditions, as well as time-variant, adaptive and non-linear systems

Transforming other expressions, e.g. discrete transfer functions or continuous expressions, into this form is relatively straightforward. A nth-order linear time-invariant system with α inputs and β outputs can be described by the next state and output equations as follows:

$$\mathbf{X}(k+1) = \mathbf{AX}(k) + \mathbf{BU}(k) \tag{4.1}$$

$$\mathbf{Y}(k) = \mathbf{CX}(k) + \mathbf{DU}(k) \tag{4.2}$$

where k represents the k-th sample instant, $\mathbf{X}(k)$ is a n-dimensional state vector, $\mathbf{Y}(k)$ is a β-dimensional output vector, $\mathbf{U}(k)$ is a α-dimensional input vector, and **A**, **B**, **C** and **D** are n x n, n x α, β x n, and β x α real coefficient matrices that describe the controller's behaviour.

In summary, equations 4.1 and 4.2 describe an iterative process that performs computation on a continuous stream of input data, i.e., input data arrive sequentially and the algorithm is executed once for every input sample and produces corresponding output values. The period between two consecutive iterations is determined by the sample rate.

The complexity of the calculation is determined by the number of multiply-accumulate (MAC) operations required to produce a new set of outputs. The number of MAC operations needed to calculate the new state variables and output values using equations 4.1 and 4.2 is:

$$N_{MAC} = (n + \alpha)(n + \beta) \tag{4.3}$$

The importance of the state-space approach is that, by defining a new set of states, any number of different representations can be generated with the same response. When used for digital controllers this flexibility can be exploited to optimise the numerical performance of the real-time equation. This point will be illustrated in the following section, in which different types and structures of digital filter will be converted into state-space equations.

4.3 DIGITAL OPERATORS

The analysis of digital controllers relies on discrete time versions of the continuous operators. The discrete version of the Laplace transform is either the Z-transform, which is associated with the *shift operator* 'z', or the γ -transform, which is associated with the *Delta operator* 'δ' [Goodwin01]. These operators allow continuous time differential equation models to be converted to discrete time difference models. Also, continuous time transfer or state space models can be converted to discrete time transfer or state space models in either the z or δ-operators. The general formulation of equations 4.1 and 4.2 can be implemented using either operator. The choice of a particular operator is largely based on preference and experience.

Despite all the advantages offered by digital controllers, there is an inherent limitation on their accuracy caused by the finite number of bits used to represent the signals. A particularly important issue when implementing a digital controller is that of the sensitivity on the filter properties to rounding errors in the representation the filter coefficients, this is known as *coefficient sensitivity.* Some filters are inherently sensitive to small changes in the coefficient values, and as a consequence, coefficient rounding errors may cause large errors in the implementation of the controller [Goodwin92].

4.3.1 *z*-operator

The *z*-operator is the most commonly used in the literature and is the traditional choice for many engineers. It is defined as:

$$z\,x[k] \triangleq x[k+1] \tag{4.4}$$

Using the z-operator, Equations 4.1 and 4.2 become:

$$\mathbf{X}_z(k+1)=\mathbf{A}_z\mathbf{X}_z(k)+\mathbf{B}_z\mathbf{U}(k) \qquad (4.5)$$

$$\mathbf{Y}(k)=\mathbf{C}_z\mathbf{X}_z(k)+\mathbf{D}_z\mathbf{U}(k) \qquad (4.6)$$

where the matrices $\mathbf{A}_z$, $\mathbf{B}_z$, $\mathbf{C}_z$ and $\mathbf{D}_z$ describe the controller when the z-operator is used and $\mathbf{X}_z(k)$ contains the controller states.

The use of the z-operator generally leads to simple expressions and emphasises the sequential nature of sampled signals [Goodwin01]. However, it presents numerical problems when used to implement digital controllers for high-speed high-performance control systems. This problem is particularly critical in recursive filters in which the sample frequency is several orders of magnitude higher that the dominant frequencies of the filter [Goodall90].

To illustrate this problem, consider the direct form II 2nd order z-filter shown in Figure 4.1.

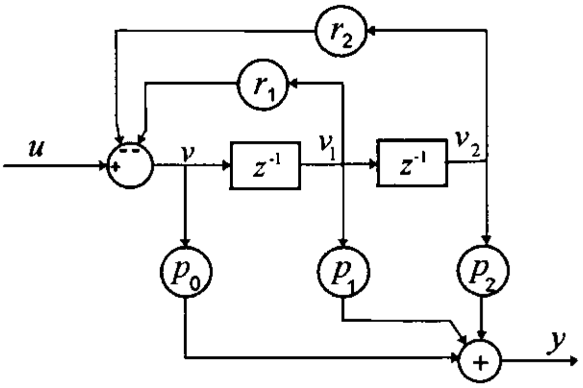


Figure 4.1 Direct form II 2nd order z-filter

The corresponding state-space representation is:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} -r_1 & -r_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u \qquad (4.7)$$

$$y = \begin{bmatrix} p_1 - r_1 p_0 & p_2 - r_2 p_0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + p_0 u \qquad (4.8)$$

Note that in practice we need to use negative rather than positive powers of z, and the corresponding equation needed to calculate the internal variable v is:

$$v(k) = u(k) - r_1 v(k-1) - r_2 v(k-2) \qquad (4.9)$$

for a real-time implementation this equation can be rewritten in the form

$$v_0 = u_0 - r_1 v_1 - r_2 v_2 \qquad (4.10)$$

The equation needed to calculate the output y is:

$$y_0 = p_0 v_0 + p_1 v_1 + p_2 v_2 \qquad (4.11)$$

When high sampling frequencies are used, the difference between successive input samples can be very small. Thus, the values of the coefficients r_1 and r_2 have to be chosen so that small differences between successive values of v can be combined to obtain the required value of y (see Equation 4.11). As a consequence, any small change in the value of any coefficient will result in a much larger change in the value of y.

4.3.2 δ -operator

It is recognised that the use of an alternative operator, namely the δ -operator, overcomes the numerical problems associated with the z-operator [Middleton90, Goodwin92].

The δ -operator is defined as

$$\delta\, x[k] \triangleq \frac{x[k+1]-x[k]}{T} \tag{4.12}$$

where T is the sample period.

From Equations 4.4 and 4.12, we can extract the relation between both operators

$$\delta = \frac{z-1}{T} \tag{4.13}$$

or

$$z = \delta T + 1 \tag{4.14}$$

Thus, any system expressed in terms of z can be converted to a model in δ and vice versa [Feuer96].

In this research we use an alternative simpler definition of the δ -operator that is more relevant when the focus is upon implementation rather than theoretical analysis [Goodall93, Forsythe91, Goodall85]:

$$\delta = z - 1 \tag{4.15}$$

Just as the z-operator, this definition is not directly implementable for real-time applications because of the positive power of z. Thus we use the inverse of δ, δ^{-1}, that is expressed in terms of z^{-1}:

$$\delta^{-1} = \frac{1}{z-1} = \frac{z^{-1}}{1-z^{-1}} \tag{4.16}$$

The δ -operator can be realised as shown in figure 4.2. The operation δ^{-1} is an accumulation, which means that the next value of w is the result of adding v to the previous value of *w*. In other words, *v* is the difference between the current and the new value of *w* (Equation 4.17).

$$w_{k+1} = w_k + v_k \tag{4.17}$$

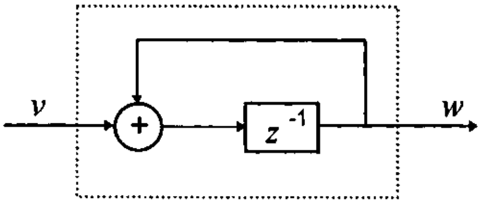


Figure 4.2 Operation δ^{-1} expressed in terms of z^{-1}

Equations 4.1 and 4.2 can also be used to represent the δ form with a different choice of controller states, and with corresponding changes in **A**, **B**, **C** and **D** matrices. Using the δ-operator, Equations 4.1 and 4.2 become:

$$\mathbf{X}_\delta(k+1) = \mathbf{A}_\delta\mathbf{X}_\delta(k) + \mathbf{B}_\delta\mathbf{U}(k) \tag{4.18}$$

$$\mathbf{Y}(k) = \mathbf{C}_\delta\mathbf{X}_\delta(k) + \mathbf{D}_\delta\mathbf{U}(k) \tag{4.19}$$

where $\mathbf{A}_\delta$, $\mathbf{B}_\delta$, $\mathbf{C}_\delta$ and $\mathbf{D}_\delta$ describe the controller when the δ-operator is used, and $\mathbf{X}_\delta$ (*k*) contains the controller states.

The δ-operator has the following characteristics:

- It emphasises the link between continuous and discrete systems, as it resembles a differentiation [Goodwin01]

- For high sample frequencies, the coefficients in $\mathbf{A}_\delta$ and $\mathbf{B}_\delta$ become almost independent of the sample period and the coefficient values closely resembles the coefficients of the corresponding continuous model [Middleton90].

- The high coefficient sensitivity problem which exists with the *z*-operator disappears completely, leaving 'normal' sensitivity in which the discrete coefficient simply need to have the same accuracy as is required for the overall performance (tipically 5% for control) [Forsythe91].

- The relation between δ and z is algebraic, thus it offers the same flexibility in the modelling of discrete time systems as the *z*-operator.

Figure 4.3 shows a diagrammatic representation of a direct form II 2nd order single-input single-output (SISO) δ-filter.

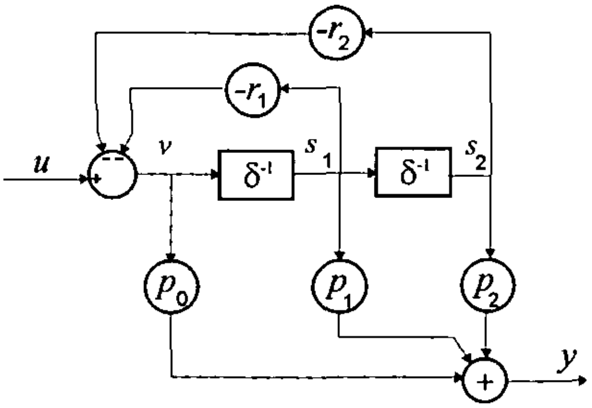


Figure 4.3 Direct form II 2nd order δ-filter

The corresponding state-space representation is:

$$\begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1-r_1 & -r_2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u \tag{4.20}$$

$$y = \begin{bmatrix} p_1 - r_1 p_0 & p_2 - r_2 p_0 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + p_0 u \tag{4.21}$$

The output y can be calculated using the following equations:

$$\begin{aligned} v &= u - r_1\delta^{-1}v - r_2\delta^{-2}v \\ &= u - r_1 s_1 - r_2 s_2 \end{aligned} \tag{4.22}$$

$$\begin{aligned} y &= p_0 v + p_1\delta^{-1}v + p_2\delta^{-2}v \\ &= p_0 v + p_1 s_1 + p_2 s_2 \end{aligned} \tag{4.23}$$

4.4 Modified controller formulation

4.4.1 Formulation description

Figure 4.4 shows a simple modification on the filter structure of Figure 4.3. In this modified form, the feedback coefficients are placed in the forward path of the filter. This modifications has the effect of scaling the state variables such as they are of similar magnitude to the input [Goodall85, Goodall93].

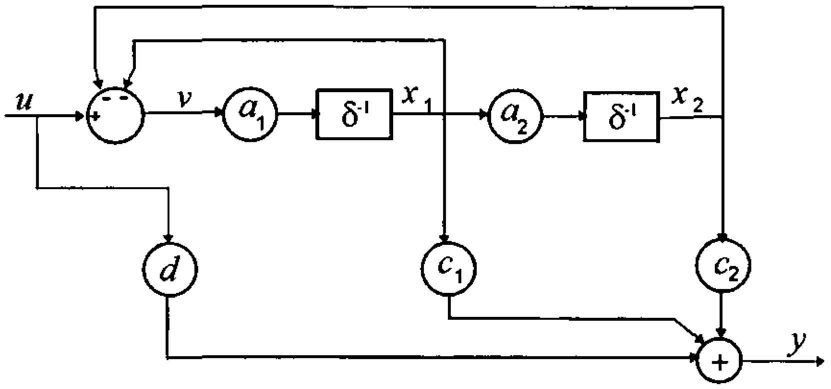


Figure 4.4 Modified form δ-filter

The corresponding state equations are:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1-a_1 & -a_1 \\ a_2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} a_1 \\ 0 \end{bmatrix} u \tag{4.24}$$

$$y = \begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + d\,u \tag{4.25}$$

The actual equations used for real-time implementation are as below; firstly the calculation of the output, then an update of the state variables so they are ready for the next sample:

$$\begin{aligned} y &= c_1 x_1 + c_2 x_2 + du \\ x_1 &= x_1 - a_1(x_1 + x_2) + a_1 u \\ x_2 &= x_2 + a_2 x_1 \end{aligned} \tag{4.26}$$

Using this modified formulation, Equations 4.1 and 4.2 can be rewritten as:

$$\mathbf{X}_{\mathrm{mod}\,\delta}(k+1) = \mathbf{A}_{\mathrm{mod}\,\delta}\mathbf{X}_{\mathrm{mod}\,\delta}(k) + \mathbf{B}_{\mathrm{mod}\,\delta}\mathbf{U}(k) \tag{4.27}$$

$$\mathbf{Y}(k) = \mathbf{C}_{\mathrm{mod}\,\delta}\mathbf{X}_{\mathrm{mod}\,\delta}(k) + \mathbf{D}_{\mathrm{mod}\,\delta}\mathbf{U}(k) \tag{4.28}$$

where $\mathbf{A}_{\mathbf{mod}\delta}$, $\mathbf{B}_{\mathbf{mod}\delta}$, $\mathbf{C}_{\mathbf{mod}\delta}$ and $\mathbf{D}_{\mathbf{mod}\delta}$ describe the controller when the modified δ form is used, and $\mathbf{X}_{\mathbf{mod}\delta}$ (k) contains the controller states.

4.4.2 Computation requirements

The general form of the state-space equations using the modified δ form is:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 1-a_a & -a_1 & -a_1 & & -a_1 & -a_1 \\ a_2 & 1 & 0 & \cdots & 0 & 0 \\ 0 & a_3 & 1 & & 0 & 0 \\ & \vdots & & \ddots & & \\ 0 & 0 & 0 & & 1 & 0 \\ 0 & 0 & 0 & \cdots & a_n & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,\alpha} \\ b_{2,1} & b_{2,2} & & b_{2,\alpha} \\ \vdots & & \ddots & \\ b_{n,1} & b_{n,2} & & b_{n,\alpha} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_\alpha \end{bmatrix} \tag{4.29}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_\beta \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & & c_{2,n} \\ \vdots & & \ddots & \\ c_{\beta,1} & c_{\beta,2} & & c_{\beta,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,\alpha} \\ d_{2,1} & d_{2,2} & & d_{2,\alpha} \\ \vdots & & \ddots & \\ d_{\beta,1} & d_{\beta,2} & & d_{\beta,\alpha} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_\alpha \end{bmatrix} \tag{4.30}$$

As Equation 4.29 shows, the modified δ form affects the **A** and **B** matrices in the state-space equations. The structure of the matrix **A** for calculating the next state variables contains a large number of 0's and 1's and has a regular structure. This allows us to reduce the total calculation requirements, because a full matrix multiplication is not longer necessary.

The number of multiply-accumulate (MAC) operations needed to calculate the new state variables and output values when Equations 4.29 and 4.30 are used is:

$$N_{MAC} = 2n + \alpha n + \beta n + \beta\alpha \tag{4.31}$$

Note that the number of MAC operations required is significantly reduced when compared with the number required to perform full matrix multiplication operations needed in Equations 4.1 and 4.2. For example, using a modest 4th order SISO filter, the number of MAC operations is reduced from 20 to 17. This may not seem a significant reduction in the number of operations, but if a larger 20th order 4-input 4-output controller is required, the number of MAC operations is reduced from 576 to 216, which makes the benefits of this formulation more evident.

4.4.2 Storage requirements

To implement the state-space equations, it is necessary to store two sets of controller states, $\mathbf{X}_k$ and $\mathbf{X}_{k+1}$. Once the new values have been calculated, they are used to replace the old values ready for the next sample period, but this can only be done when all values have been calculated.

A simpler solution is to overwrite the old values with new values as they are calculated, which means that only one set of controller states needs to be stored. To achieve this, it is possible to reverse the order of calculating the states.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 1 & a_1 & 0 & & 0 & 0 \\ 0 & 1 & a_2 & \cdots & 0 & 0 \\ 0 & 0 & 1 & & 0 & 0 \\ & \vdots & & \ddots & & \\ 0 & 0 & 0 & & 1 & a_{n-1} \\ -a_n & -a_n & -a_n & \cdots & -a_n & 1-a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,\alpha} \\ b_{2,1} & b_{2,2} & & b_{2,\alpha} \\ \vdots & & \ddots & \\ b_{n,1} & b_{n,2} & & b_{n,\alpha} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_\alpha \end{bmatrix} \tag{4.32}$$

The corresponding real-time equations are:

$$\begin{aligned} x_{1,k+1} &= x_{1,k} + a_1 x_{2,k} + \sum_{i=1}^{\alpha} b_{1,i} u_{i,k} \\ x_{2,k+1} &= x_{2,k} + a_2 x_{3,k} + \sum_{i=1}^{\alpha} b_{2,i} u_{i,k} \\ x_{3,k+1} &= x_{3,k} + a_3 x_{4,k} + \sum_{i=1}^{\alpha} b_{3,i} u_{i,k} \\ \vdots \quad & \quad \vdots \qquad\quad \vdots \qquad\quad \vdots \\ x_{n,k+1} &= x_{n,k} - a_n \left(x_{1,k} + x_{2,k} + x_{3,k} + \cdots + x_{n,k} \right) + \sum_{i=1}^{\alpha} b_{n,i} u_{i,k} \end{aligned} \tag{4.33}$$

By inspection it can be seen that the old values of the controller states are only needed to update the value of x_n. Thus, if a new variable σ is used to store the sum of the old values of x_1 to x_n, as soon as the are available, it is possible to avoid retaining old values for the states while the new values are calculated.

The equation used to update the value of x_n can be rewritten as:

$$x_{n,k+1} = x_{n,k} + a_n \sigma_k + \sum_{i=1}^{\alpha} b_{n,i} u_{i,k} \tag{4.34}$$

where σ_k is defined as:

$$\sigma_k = -\left(x_{1,k} + x_{2,k} + x_{3,k} + \cdots + x_{n,k}\right) \tag{4.35}$$

Thus, in practice we only need to store one set of state variables **X**, which both reduces the overall space requirements for the CSP and simplifies the operation because it is not necessary to transfer the values in $\mathbf{X}_{k+1}$ to $\mathbf{X}_k$ and the end of each algorithm cycle. It is interesting to consider that the reversed order formulation preserves a regular structure and no modification to the coefficient values are required.

From Equations 4.29 and 4.30 it is possible to calculate the number of coefficients and state variables required to implement the algorithm. The number of coefficients is shown in Table 4.1, and the number of state variables is shown in Table 4.2.

Coefficients description	**No of values**
Coefficient matrix A	n
Coefficient matrix B	$n\alpha$
Coefficient matrix C	βn
Coefficient matrix D	$\beta\alpha$
Total:	$n + n\alpha + \beta n + \beta\alpha$

Table 4.1 Number of values in coefficient format

Variables description	**No of values**
Input values	α
Output values	β
State variables	n
σ value	1
Total:	$\alpha+\beta+n+1$

Table 4.2 Number of values is state variable format

4.4.3 Summary of modified δ formulation

In summary, the advantages of using the modified δ form are:

- It minimises coefficient sensitivity. The percentage of accuracy required for the coefficients is the same, as that required for the overall characteristics of the controller. This is in contrast to the *z*-operator formulations, for which the coefficients often need to be hundreds of times as accurate [Forsythe91].

- Preserves a structured A matrix with large number of 0's and 1's, which makes full matrix multiplication unnecessary.

- Internal variables are all well-scaled. They all have the same nominal maximum values, which are of the same magnitude as that of the input.

- Avoids the need to convert the controller into cascaded 1st/2nd order sections, something that is almost essential for high-order controllers using formulations based upon the *z*-operator [Forsythe91].

- It is superior numerically to any structure based upon the *z*-operator, particularly at fast sampling rates.

The possibility of using either operator provides a design freedom that the designer can use; for a given control law the most appropriate set of states can be freely chosen to optimise the controller formulation from the numerical point of view. However it is important to appreciate that these are only optimal in the strict mathematical sense because generally the controller **A** matrix will be fully populated with elements, for each of which a multiplication is required. Other formulations may be strictly sub-optimal by comparison, but many of the A matrix elements will be 0 or 1. If the full matrix equation is calculated this makes no difference, but if the structure of the **A** matrix is recognised it is possible to extract the essential equations from the full matrix formulation and thereby reduce considerably the number of computations which are needed.

4.5 Numerical representations

The accuracy and dynamic range of coefficients and state variables directly influence the overall system response. When the digital controller is implemented in a general-purpose processor, it may be sufficient to find coefficient and state variable word lengths that can be represented with the data types provided by the processor and that satisfy the system response requirements.

Very high sample frequencies result in long word length requirements for both coefficients and state variables. This is because the difference between successive values of the input and output become increasingly small. The δ -operator avoids some of the problems especially with respect to the coefficients [Forsythe91]. However, the variable's word lengths need to be carefully chosen to ensure that the full value and dynamic range of the variables involved in the calculation can be accommodated.

Although it is possible to select arbitrarily large word lengths when the controller is implemented using flexible processing elements or dedicated hardware, it is important to keep them to a minimum. The selected word lengths will determine the size of the arithmetic blocks, which has a major impact on the amount of hardware resources, maximum speed, and power consumption.

4.5.1 Coefficient format

The low coefficient sensitivity of the formulation allows the use of short word lengths to represent the coefficients. Figure 4.5 shows a general format for the coefficients. They are held in a simple low-precision floating-point form, with a 6-bit mantissa in two's complement format and a 5-bit exponent. The position of the binary point in the mantissa is predetermined to allow for fractional values. This is because the coefficients are always less than unity, with values that become progressively smaller as the sample frequency is increased. The exponent has a biased range of +6 to -25. The positive end of the exponent range is provided to implement gains greater than unity, which is common in control systems. On the other side, a negative exponent value allows to represent values that are significantly smaller that unity, which is a characteristic of the controller formulation based in the δ-operator. This format will allow representing any coefficient with an accuracy of 1%, which is more that enough for most control applications [Forsythe91, Goodall92].

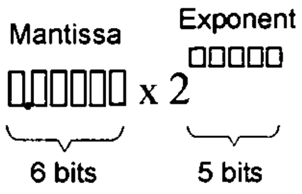


Figure 4.5 Coefficient format

4.5.2 State variable format

The use of the modified form described in Section 4.4 has the effect of scaling the state variables such as they are of similar size to the input values. This allows the use of a fixed-point format to represent the state variables. The overall bit resolution required is determined by the number of bits used to sample the input data and the number of bits required to handle internal underflow and overflow.

The number of bits per sample is determined by the of the analogue-digital and digital-analogue conversion hardware used. As described in Section 3.5, a word length of 12 bits has been adopted such that the controller can be used for general application.

Section 4.4 explained that the state variables are of similar size to the input. However, 3 overflow bits are provided in order to ensure correct operation and to reduce the number of overflow checks.

The state values are formed by an accumulation of small values. Thus, underflow bits are provided to ensure that small input values will not be truncated when multiplied by a small coefficient value and their effect will propagate through the controller. The number of underflow bits can be derived from the structure in terms of coefficients and the required fractional output accuracy for small inputs.

A simple criterion used to determine the number of underflow bits is described in [Goodall85]. Consider the filter shown in Figure 4.4. It is desired that x_2 responds to a one least significant bit change in v. In order that x_2 changes it must have a resolution of $1/a_2$ bits (i.e. $\log_2 (1/a_2)$) and following the same criterion, x_1 should have a resolution of $1/a_2a_1$. This can be extended for larger filter following the same pattern. A reasonable number of underflow bits would in the range of 8-16 bits, which will allow to support a wide range of controllers.

The state variable format can also be used to represent the input and outputs values of the controller. This will permit common procedures to be used to perform arithmetic operation, such as multiplication and addition. Thus, the position of the binary point is chosen so the input/output values map directly into the state variable format. The input values are brought in as signed integer form in two's complement format. The input value is sign-extended to the left to fill the overflow bits and the underflow bits are set to zero. Figure 4.6 shows the general state variable format.

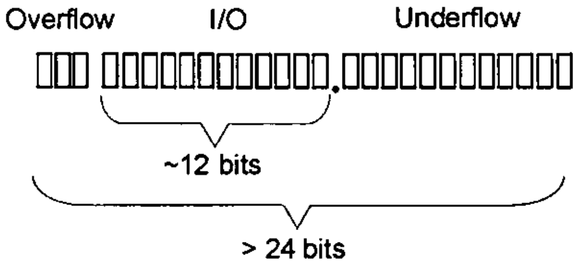


Figure 4.6 State variable format

The adoption of these particular formats offers a significant reduction in computation time by avoiding the need of complex operations using standard floating-point formats. Of course if there are exceptional requirements it is always possible to redefine the formats, maintaining the essential principles but extending the precision as required by a particular application, although the test results later will show that they are sufficient even for extremely demanding applications.

4.6 SUMMARY OF THE CHAPTER

This chapter has described a controller formulation that is suitable to be efficiently implemented in hardware. It identified the state-space models of control systems as specially suited to implementations using computer solutions because the number of functions that are required is quite limited and specific.

The numerical problems associated with z-operator when implemented at fast sampling rates were identified, and it was pointed that the use of the δ-operator overcomes a number of those problems.

From a numerical point of view it is preferable to use the δ -operator rather than the z-operator when implementing discrete transfer functions. This is because it offers a more robust implementation and improved performance under similar implementation constraints, such as finite word length for the filter coefficients and state variables. It is recognised that the use of the δ-operator overcomes the numerical problems associated with the z-operator [Middleton90, Goodwin92]. A study that shows the superiority of the δ-operator over the z-operator is found in [Forsythe91] and [Goodall93].

To conclude this chapter, we can identify a number of properties of the algorithm that facilitate its hardware implementation:

- A set of inputs is used to produce a set of outputs within each sample period
- There are no data dependent operations
- The algorithm loop must be executed continuously

Chapter 5

CSP hardware implementation

5.1 OBJECTIVES OF THE CHAPTER

This chapter develops a hardware implementation alternative that is suitable for the controller formulation described in chapter 4. The proposed architecture takes advantage of the analysis of the specific control application to permit efficient and cost effective realisations of the required processing functions. The objectives of this chapter are:

- To describe the process used to map the control algorithm directly onto a hardware structure
- To provide an analysis of the CSP core
- To explain the resulting CSP architecture and its external interface
- To present the results of synthesising this architecture

5.2 MAPPING THE CONTROL ALGORITHM INTO HARDWARE

The match between the architecture of the processor and the structure of the algorithm that we wish to implement on this processor will determine the efficiency with which the algorithm is executed. The efficiency of the implementation can be measured in terms of speed, cost and power consumption. The main goal is to design an architecture that matches the algorithm and not vice versa. This implies choosing an interconnection scheme for the various hardware entities that will allow efficient execution of the instruction set we chose, preferably executing one instruction per clock cycle. It also implies designing an instruction set that best performs the type of operations required. The purpose of the mapping process is to determine the type and number of processing elements (PE), the size and number of the memories, and the required communications channels.

5.2.1 Software model

A model of the CSP processor that implements the control algorithm defined in Section 4.4 is programmed in Java. This model is used to validate the correctness of the control algorithm to be implemented. The validation is done by comparing the results obtained by the model against the results of a Matlab program, which uses 32-bit floating-point variables to perform the calculations. A correct analysis of the controller requirements should ensure that the CSP's behaviour is satisfactory. If the results obtained by the model are not satisfactory, then it is necessary to increase the resolution of the state variables and/or coefficients to achieve the precision required. Thus, the software model can be seen as a functional specification of the processor.

Another function of the software model is to produce test vectors that can be used to verify the hardware model of the CSP. Test vectors can be created for each functional block of the CSP, as well as for the whole processor. The test vector files are created during the execution of the software model. These files contain the inputs and expected outputs of each block stored in text format.

5.2.2 Mapping process

To map the algorithm to a hardware structure, the algorithm is divided into tasks or processes. These processes are data storage, instruction fetching and decoding, next instruction address calculation, and arithmetic operations. This partitioning should allow easy mapping of the processes into hardware structures. The main goal during this process should be to minimise the resources required. This process is simplified by the fact that the algorithm mainly involves straightforward arithmetic operations.

The number of concurrent operations can determine the amount and functionality of the hardware structures. For example, the maximum number of simultaneous memory transactions determines the number of memory ports required and therefore the number of communication channels between the processing elements and memories.

The execution of the control algorithm requires the repeated execution of a set of instructions. Because the number of instructions in the control loop can be small when implementing simple controllers, the overhead imposed by the instructions that manipulate the program counter may be relatively large [Lapsley97]. Thus, special attention must be given to a control structure that implements loops of instructions. Thus, the CSP should provide a looping mechanism that introduces a short, or ideally, zero overhead.

The final step is to create a hardware model that supports the operations needed to implement the algorithm. This hardware model is programmed using the hardware specification language VHDL. The hardware model is then simulated and verified using as reference the test vectors produced by the software model. Finally, the model is synthesised, and the netlists downloaded into the programmable device that will be used to perform a final verification.

5.2.3 Architecture options

In this section we discuss the factors that affect the selection of an appropriate architectural structure for the control system processor. The motivation to search for a specialised structure lies in the fact that the control algorithms to be performed are well defined and, as described in the summary of Chapter 4, present characteristics that can be exploited to achieve efficient execution.

At the architectural level, the main interest is to look at the general organisation of the system, this involves the definition of components such as processing elements and memories, and the specification of interfaces and control strategy. The design of the system architecture is affected by several parameters such as memory capacity, access time, word length, and processing times.

While designing the system architecture, we must be able to estimate the system performance to identify and correct potential bottlenecks. The basic idea is to design an architecture where processing elements execute operations indicated by a program, and that the processing elements are supported by appropriate communication channels and memories.

There are three basic approaches to implement the CSP.

- At one extreme, a single PE executes all the arithmetic operations. The PE must be able to execute all the operations. The processing time will equal to the product of the PE processing time and the total number of operations. This approach is power and area efficient.

- At the other extreme, one dedicated PE is assigned to each basic operation. The PE can therefore be optimised to execute a specific operation. The maximum number of PEs is determined by the parallelism in the algorithm. The slowest PE as well as data availability determines the maximum sample rate. This approach leads to a high throughput at the expense of large power consumption and chip area.

- An intermediate solution that combines the two approaches described above.

The first approach will be used to implement the CSP, mainly because our aim is to produce an architecture that uses a minimum amount of hardware resources. Furthermore, when implementing the control algorithm using a parallel architecture, the size and performance of the architecture depend on the system characteristics and can be difficult to predict, especially for large systems.

5.3 PROCESSING ELEMENT

Now that we have chosen an approach to implement the processor, we need to optimise it according to the systems requirements. Processing elements usually perform simple operations that map the input values to a single output value. Normally, they do not have storage capabilities and ideally perform the operation in a single clock cycle.

5.3.1 MAC unit

The most obvious processing elements to implement the sum of products (Equation 2.1) required by the algorithm is multiply-and-accumulate (MAC) (Figure 5.1).

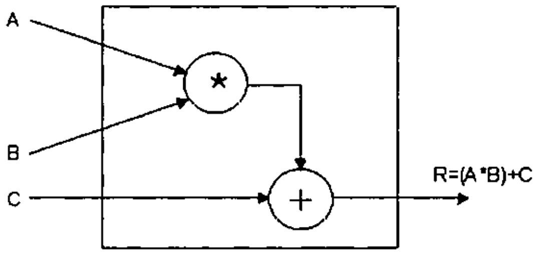


Figure 5.1 Processing element

The MAC unit performs the operation D=A*B+C, where A is a coefficient and B and C are state variables. To perform the multiplication A*B, the coefficient has to be divided into its mantissa and exponent parts. The mantissa is multiplied by the state variable B and the result is shifted according the exponent value. Finally, to complete the MAC operation, this result of the multiplication (already in state variable format) is added to the state variable C (Figure 5.2). The following sections describe the MAC unit components. Simulation and synthesis results are presented in Section 5.8.

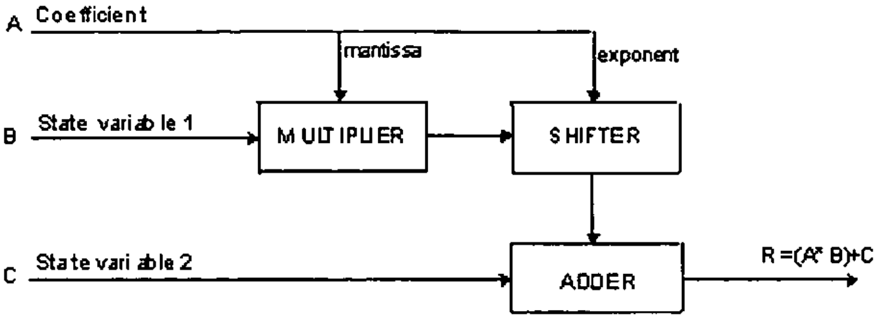


Figure 5.2 MAC unit

5.3.2 Array multiplier

The presence of a single-cycle multiplier is essential to achieve high performance for the CSP. This is because most of the operations performed by the processor involve a multiplication. There are many options to implement hardware multipliers. The multiplier selected for the CSP uses the Baugh-Wooley technique [Baugh73] [Pirsch98]. It multiplies two numbers in two's complement format. Figure 5.3 shows a block diagram of the array multiplier. The partial products are formed by an array of AND gates. These partial products are then added together to produce the results. To perform the addition, the multiplier includes several carry-save adders (CSA) and one carry lookahead adder (CLA).

Unlike carry propagate adders that evaluate the carries to determine the sum result. The idea of the carry-save adder is to 'save' the carry for the next stage. This means that the carry signals are not used for the current addition, but rather for the successive adders. A CSA consists of an array of full adders that merge three operands to one sum and one carry values.

The number of CSA adders depends on the number of bits of the coefficient's mantissa. In Figure 5.3, the four CSA adders reduce the number of operands from six to two, which are then added by the CLA to produce the multiplication result. The carry lookahead adder consists of a series of full adders and, as its name indicates, also includes parallel logic to evaluate the carries, which speeds up the addition process. This compact and regular structure results in an efficient and fast multiplier.

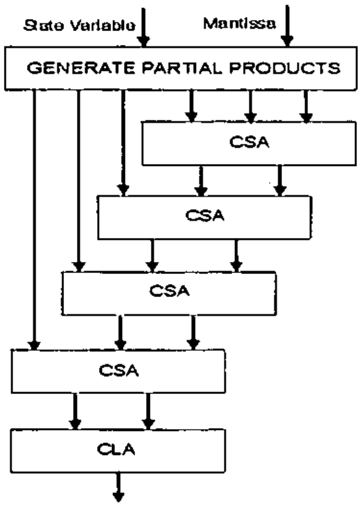


Figure 5.3 Block diagram of the array multiplier

5.3.3 Shifter

A shifter is placed immediately after the array multiplier to complete the multiplication process. Traditional shifters offer a left shift by one, a right shift by one, or no shift. Such shifters can also perform multibit shifts, but this is done one bit at a time and can be time consuming. Another kind of shifter, called barrel shifter, offers more flexibility by supporting shifts by any number of bits in a single cycle. Because, the input has to be shifted according to the exponent value, which has a biased range of -25 to +6 (see Section 4.5.1), a specially adapted barrel shifter in required. A 6-bit positive shift (left shift) indicated by a exponent value 31 will scale the input value by 2^6, while a 25-bit negative shift (right shift) indicated by an exponent value 0 will scale the input by 2^{25}. A no shift is indicated by an exponent value 25.

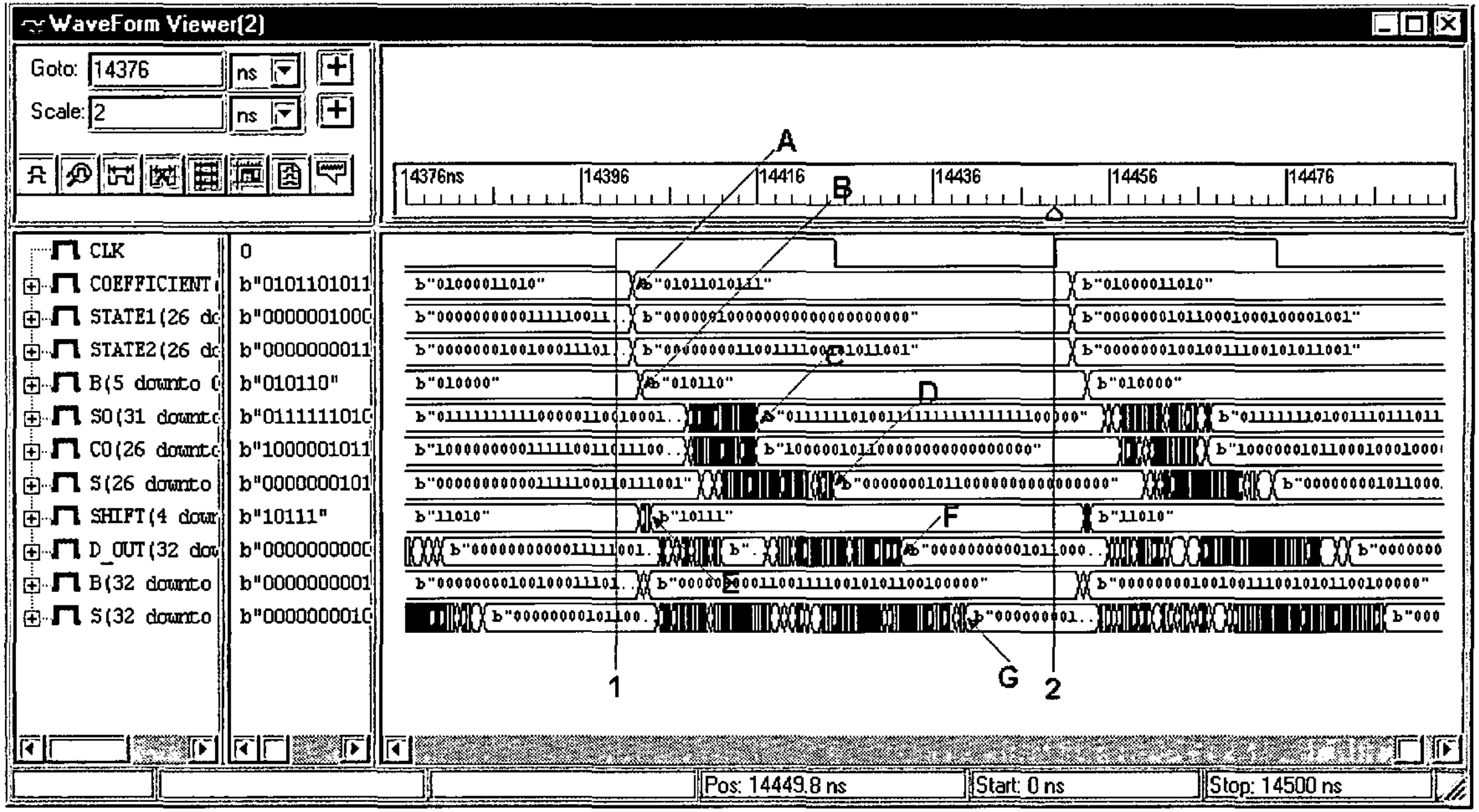
A shift to the right duplicates the sign bit (either a one or zero) into the most significant bits (arithmetic shift). A shift to the left inserts zeros into the least significant bits. When shifting a value to the left, the shifter checks for overflow. If a positive overflow occurs, the output is set to the maximum positive value. While for a negative overflow, the output is set to the maximum negative value.

5.3.4 Adder

The shifter just described is used to complete the multiplication of the coefficient and one of the state variables. The multiplication result, which is already in state variable format, is added to the second state variable to complete the multiply-accumulate operation. A CLA adder, like the one described as part of the array multiplier, is used to perform this final addition.

5.3.5 MAC unit simulation

To illustrate the MAC unit functionality, consider the waveform graph shown in Figure 5.4. The MAC operation is performed in a single clock cycle. The input data that was read from the data memory at the rising edge 1 is passed to the MAC unit (A). The coefficient is then divided into its mantissa and exponent parts (B and E respectively). The state variable 1 and the mantissa are used to feed the array of CSA adders to produce sum and carry vectors (C) that are added to complete the multiplication (D). The result of the multiplication is then shifted according to the value of the exponent (E). And finally, the output of the shifter (F) is added to the state variable 2 to obtain the result of the MAC operation (G).



Goto: 14376 ns

Scale: 2 ns

Pos: 14449.8 ns

Start: 0 ns

Stop: 14500 ns

Figure 5.4 MAC unit simulation waveform

5.4 MEMORY SYSTEM

The MAC unit has been optimised to provide high performance multiply-accumulate operations. Because one of the requirements to implement the CSP is that it should be able to execute one MAC operation per clock cycle, the memory system must allow the CSP to fetch an instruction while simultaneously fetching operands for the instructions and storing the result of the previous instruction. This involves the completion of complete several accesses to memory simultaneously. Thus, the organisation of memory and its interconnection with the MAC unit are critical to achieve high performance for the CSP.

5.4.1 Memory architecture

The memory elements store data so that the PE, in addition to implementing the algorithm, can access appropriate data without loss of any computational time slots. Since the PE requires several simultaneous inputs and outputs, we require that the memories be partitioned into several independent memories, or have several ports, which can be accessed in parallel.

To achieve the required performance, the memory system must allow the CSP to perform the following processes within one instruction cycle:

- Fetch the instruction to be executed
- Read the appropriate operands
- Write the result of the previous operation

This means that the processor must make five accesses to memory in one instruction cycle (4-read, 1-write).

The simplest option is to have single bank of memory where all the instructions and data will be stored. However, the number of bits used to represent the coefficients is different to the number of bits in the state variables and instructions. This means that the memory must be wide enough to allocate the widest of them and in such case, entries used to store shorter words will be underutilised. To avoid this problem and to provide the ability of accessing instructions and data simultaneously, three independent memories will be used in the CSP: program memory, multiport data memory (three read, one write), and initialisation data memory.

5.4.2 Data memory

The data memory contains all the data required to perform the algorithm, namely: coefficients, state variables, input and output values, and partial products. A multiported memory is used to provide the three read and one write accesses to data needed for a MAC operation. The data memory has four independent sets of address and data connections, allowing independent memory accesses to proceed in parallel.

As seen in Section 3.4.2, the A500K130 devices allows the implementation of multiported memories. Figure 5.5 shows the multiported data memory. The width of the memory blocks will depend on the number of bits used to represent the coefficients and state variables (whichever is wider). While the complexity of the controller will determine the number of total values required for the algorithm and therefore the depth of the required memory.

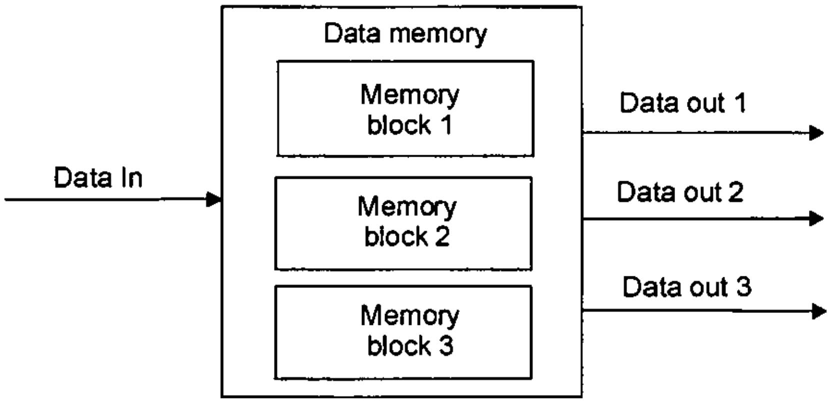


Figure 5.5 Data memory organisation

5.4.3 Program and initial data memories

The program memory contains the program that will implement the control algorithm. While the initialisation data memory contains the coefficients and initial state values needed to initialise the processor before performing the control algorithm.

For simplicity, the program and initial data memories are implemented on-chip. This is possible because the ProASIC devices offer the possibility of implementing hardwired memories using the logic tiles. This effectively creates on-chip non-volatile ROM memories that can be programmed together with the rest of the CSP components. The overall architecture is simplified, as external memory interfaces are not required.

5.4.4 Mapping input values into data memory

Input sample values must be mapped to fit within the state variable format. As both types of variables are represented using two's complement format, the mapping is straightforward. Firstly, it is necessary to copy the most significant bit of the input value to every bit of the overflow part of the state variable. Then the input value is then copied to the integer part. Finally, 0's are inserted in the underflow part to complete the state variable word. To extract an output value from the state variable format, the 12 least significant bits of the integer part must be read (Figure 5.6).

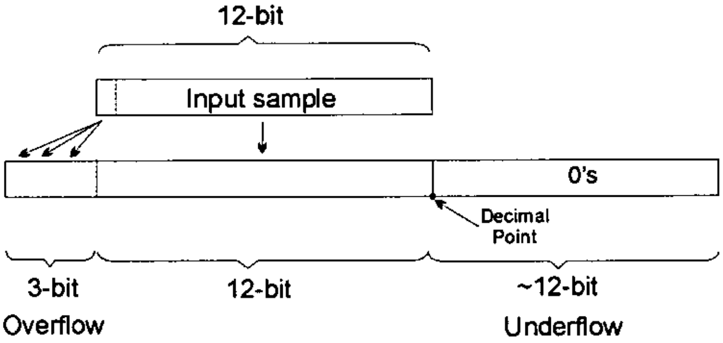


Figure 5.6 Mapping an input sample into state variable format

The output values produced by the CSP, which are extracted from an internal variable in state variable format, have been rounded to the nearest integer. This means that if the value of the underflow bits is greater than or equal to 0.5, the output value will be increased by 1.

5.4.5 Data memory organisation

Figure 5.8 shows how the coefficients and state variables can be grouped when stored in the data memory. Although the coefficients and other variable can be stored in any location in the data memory and the order shown in the figure does not necessarily has to be followed when implementing a controller, it is important to keep consistency so the design can be verified more easily.

As Figure 5.7 shows, constant values are included in both, coefficient and state variable format. These constants can be included to add flexibility to the operation the CSP can perform. The number and value of these constants will depend upon the operations required to implement the algorithm. Further explanation including some example instructions is given in Chapter 6.

5.4.6 Addressing mode

The number and position of the variables involved in the CSP algorithm remain constant during the program execution. This is because the partial products and other auxiliary values needed to perform the algorithm are stored in predetermined memory locations and kept in the same location when updated. Also, the coefficients do not change during the program execution, in fact they can be considered as constants. And finally, when a state variable is updated, the new value is rewritten on the same location.

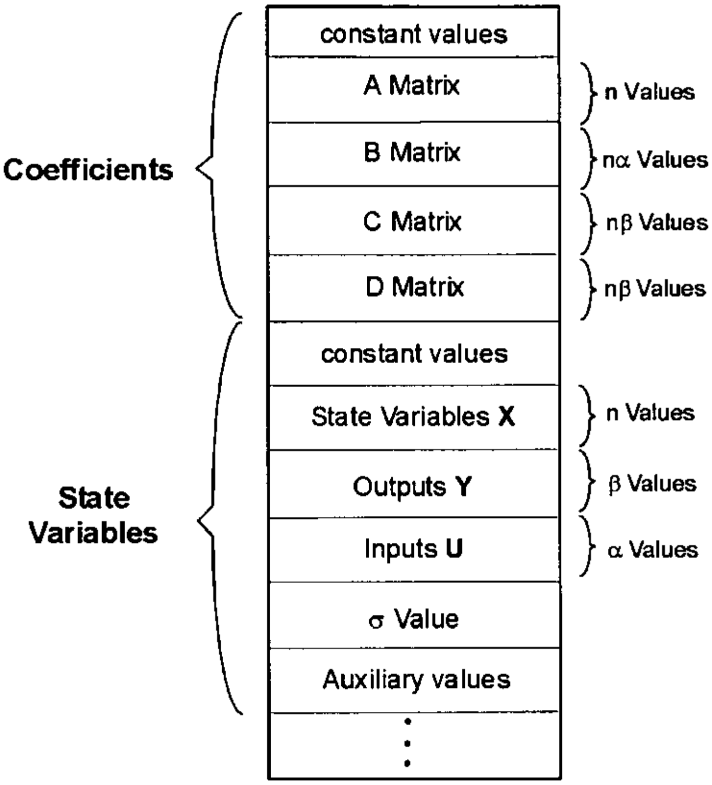


Figure 5.7 Data memory organisation

Thus it is possible to use a simple direct addressing mode to access the values stored in the memories. The addresses specified in the instruction point directly to the physical location of the variables. The number of bits needed to address the memory locations depends on the values of n, α, and β. As an example, consider a 4th-order single-input single-output controller ($n = 4$, $\alpha = 1$, $\beta = 1$), the number of coefficients and state variables are 16 and 11 respectively. The number of bits needed to address data in the memory is 5, which will allow us to address up to 32 memory locations.

5.5 CONTROL

Control strategy is mainly concerned with the manner in which control signals direct the data flow in the system. The simple sequence of operations required to implement the algorithm allows the use of a simple centralised control scheme based on an instruction handler and a program counter.

5.5.1 Program counter

Before the CSP can execute an instruction, the instruction must first be read from the program memory and brought to the instruction handler. The program counter contains the address of the next instruction in memory to be executed. The process of fetching an instruction begins with the value of the program counter being used as address to access the program memory. Once the instruction has been read, the value of the program counter is incremented by 1. In this way, the program counter indicates the next instruction while the current instruction is being executed.

To perform its task, the program counter (PC) performs uses three values: program counter, initial address, and final address. The PC value points to the address of the next instruction. At the beginning of the algorithm execution, the program counter is initialised to zero by a reset signal and automatically increased by one after each instruction is executed. When the PC value reaches the final instruction addresses, the initial instruction address is copied to program counter. This procedure creates an loop to process the inputs and generates the outputs of the CSP. Figure 5.8 shows the program counter algorithm and Figure 5.9 shows how the algorithm is implemented in hardware.

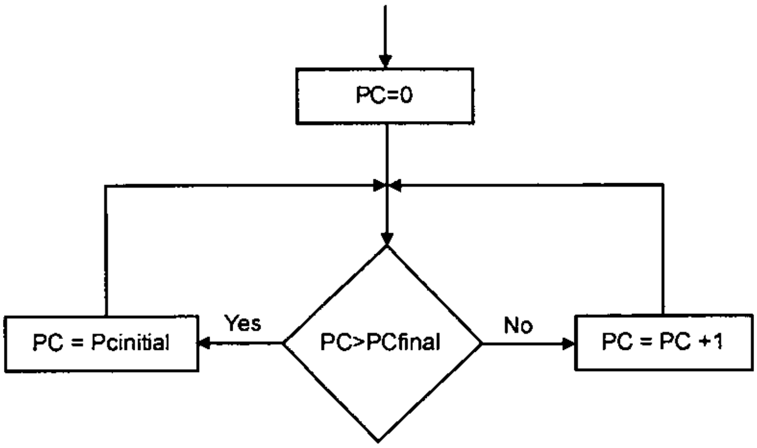


Figure 5.8 Program counter algorithm

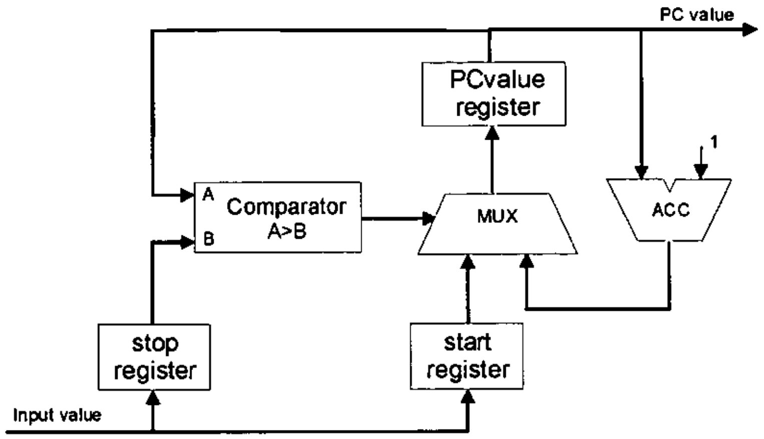


Figure 5.9 Program counter hardware

5.5.2 Instruction handler

The instruction handler decodes the instruction to be executed. It divides the instruction into fields. The operation code field indicates what CSP instruction is to be executed. The other fields contain the address of the data to be used by the instruction. The instruction handler generates signals to control the CSP operation according to the operation code read from the instruction. It also extracts the source and destination addresses and controls memory accesses by enabling read and write signals to the memories.

5.6 CSP ARCHITECTURE

Figure 5.10 shows a block diagram for the CSP system. The core of the CSP comprises the MAC unit and the data memory block. This architecture employs separate program and data buses to access separate data and program memories, an arrangement that increases speed since instructions and data can move in parallel and execute simultaneously rather than sequentially. The computation of the output values is done by iteratively executing multiply-accumulation (MAC) operations.

The following steps are needed to complete an instruction execution cycle.

1. The value of the program counter is transferred to the read address of the program memory

2. The content of the specified memory locations is transferred to the instruction handler

3. The instruction handler decodes the instructions. It places the appropriate memory addresses and control signals to control the data flow through the processor

4. The program counter value is updated and the process repeats for the next instruction

The process of reading and decoding the instruction and the pipeline stage produce a latency of 4 clock cycles between instructions issues and the result being written back to the data memory.

The CSP architecture has also been designed to allow for 'block-structured' controllers, in which a number of transfer-function blocks, each implemented by the formulation described in Section 4.3, can be arbitrarily interconnected from inputs to outputs. This approach can be used to reduce controller complexity if it is possible to identify appropriate sub-structures.

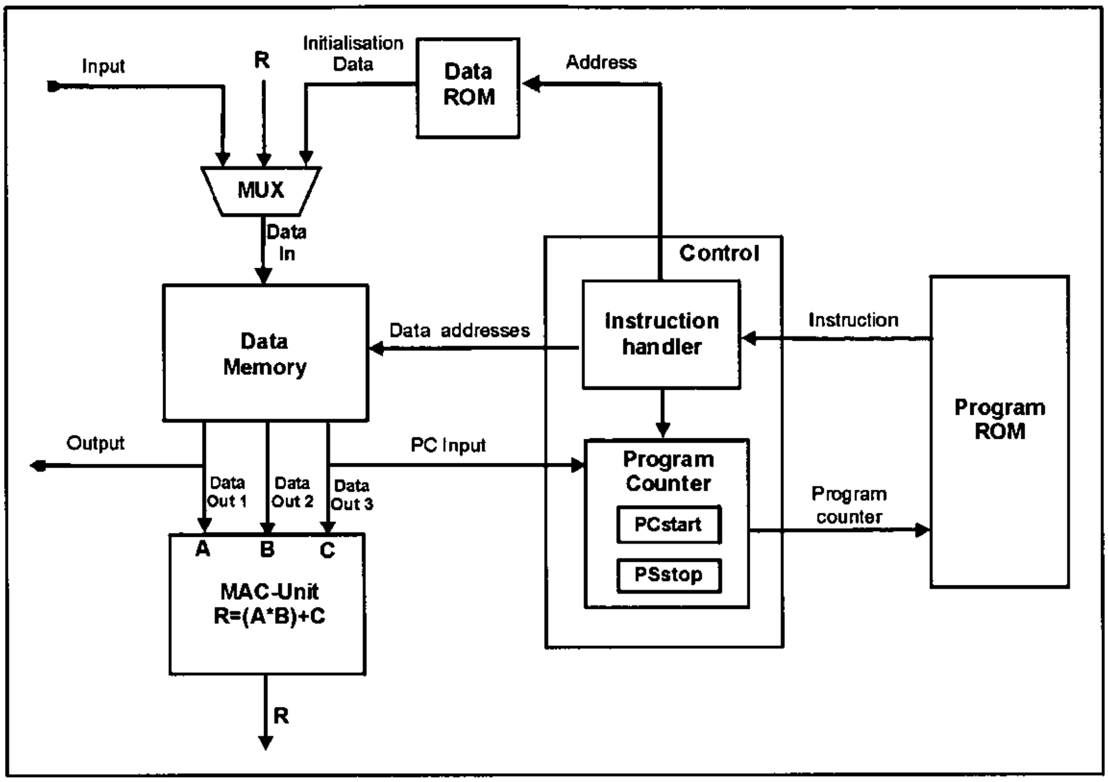


Figure 5.10 CSP block diagram

5.7 PIPELINING

Pipelining is commonly used to speed up a processor by breaking the execution of instruction into smaller processes and executing these processes in parallel if possible. Thus, decreasing the time required to execute a sequence of instructions. Strictly speaking, the CSP architecture is pipelined as it performs the following tasks in parallel:

- Fetch a new instruction from program memory
- Decode the instruction
- Retrieve data operands from data memory
- Execute the operation

Although the MAC unit can produce one result per clock cycle, the internal pipelining results in a delay (or latency) of four cycles from the time the instruction is fetched from the program memory until the result is available at its output. This latency can result in data dependency if the result of one instruction is needed by the following instruction. The data dependency problem and how it can be solved is explained in Chapter 6.

5.8 HARDWARE COMPLEXITY AND CLOCK SPEED

5.8.1 Parameters used for hardware implementation

The CSP design used to obtain the results shown in this section was implemented using the following parameters.

- Coefficients format: Low-precision floating-point form, with a 6- bit mantissa in two's complement format and a 5-bit exponent.

- State variable format: 27-bit signed form in two's complement format.

- I/O data format: 12-bit signed integer form in two's complement format.

- Data memory: 27-bit x 256 word 3-read 1-write RAM.

The CSP implements a 4th order single-input single-output controller. A full description of the structure of the CSP program is presented in Section 6.3.1 and a description of the controller example in Section 7.3.1.

5.8.2 Synthesis results

Table 5.1 shows the CSP complexity in terms of Actel's ProAsic device tiles and equivalent gates [Actel00a]. Everything except the program and data memories are fixed in size; these memories are hardwired, and their size and speed depends upon the control algorithm being implemented. The figures shown are for the fourth-order single-input single-output filter described in Section 4.3.

The synthesis of the CSP core results in an overall gate count of 1560 ProASIC logic tiles, which is equivalent to approximately 12000 system gates. The program and data ROM memories requires 980 logic tiles overall. The data memory block is implemented using 9 embedded RAM blocks provided by ProAsic devices.

The relatively small size of the processor core leaves much of the FPGA free such that it can be used to carry out additional functions defined by the user. In this case, the CSP used about 20% of the area available in an A500K130 device, which is a medium range device of the A500K family.

Block	**ProAsic Tiles**	**Equivalent gates**
Instruction Handler	101	808
MAC Unit	1105	8840
Program counter	175	1400
I/O Block	60	480
Pipeline registers	120	960
Program ROM	900	7200
Data ROM	80	640
Total	2541	20328

Table 5.1 CSP complexity

Table 5.2 shows the delay information for the CSP produced at different stages of the design flow. The first column shows the CSP parts, while the rest of the column shows the delay information for each part. The second column shows the delay information produced by the synthesis tool (Leonardo Express). The third column shows the delay information produced by ASICMaster after place and route and the third column shows the delay indicated by the static timing analyser (Flash Timer).

The delay information produced by the different tools can vary considerably. For simplicity and following a recommendation from Actel, we will base our analysis on the results produced by the timing analyser.

Module	Synthesis	Place and route	Timing analyser
MAC	75.61	61.64	76.41
Instruction Handler	12.67	10.41	12.45
Program counter	21.65	21.46	26.44
Program ROM	55.25	53.34	64.92
Data ROM	15.05	12.80	14.38

Table 5.2 CSP delay information (ns)

It can be seen form Table 5.2 that the MAC unit and program ROM are the slowest parts of the design. As seen in Section 5.4.3, the program ROM is implemented using logic tiles, thus its size and delay depend on the CSP program to be implemented. For that reason, the placement and timing restrictions used to optimise one design may not produce good results for other designs. In fact, for larger controllers, the program ROM may contain the critical path of the design. However, it is important the remark that this problem is a consequence of the method used to implement the ROM and can be solved by using an external ROM to hold the CSP program.

Unlike the program ROM, the MAC unit has a regular structure that can be exploited to improve its performance. The execution time of the MAC unit can be accelerated by the introduction of a pipeline stage. Ideally, the pipeline should be placed so it divides the MAC data path into two parts with similar delay. To identify the best location for the pipelined within the MAC unit data path, each block of the MAC unit was modelled in VHDL and synthesised individually. The multiplier was divided into its CSA and CLA parts, thus splitting the data path within the MAC into four sections: multiplier's CSA and CLA sections, barrel shifter, and CLA adder. A detailed low level design has been used to speed up each part of the MAC operation. This involves the introduction of a number of placement and timing constraints prior to placing and routing individual blocks in order to have the minimum possible delay between logic elements. To have an accurate estimate of the delays; registers were placed at the inputs and outputs of each block before doing the synthesis. Tables 5.3 to 5.6 show the results obtained after synthesising the different blocks.

Tables 5.3 and 5.4 show the delays of the multiplier's CSA and CLA adders respectively. Table 5.5 shows the delay of the complete multiplier and Table 5.6 shows the delay of the barrel shifter. From these table it can be seen that the delay of each of the four parts of the MAC unit is very similar to each other. Thus the most suitable position for the pipeline is located at the output of the multiplier as shown in Figure 5.11. This effectively divides the MAC unit into two parts that have similar delays, about 43ns for the multiplier and 39ns for the shifter and CLA adder. The delays shown in Table 5.5 were obtained by synthesising the CSA and CLA adders combined into a single block rather than just adding the delays of the CSA and CLA adders shown in Table 5.3 and Table 5.4 respectively

Module	Synthesis	Place and route	Timing analyser
Array of CSA adders	21.15	18.48	23.32
Array of CSA adders with place & route constraints	21.15	16.31	20.55

Table 5.3 CSA adder delay information

Module	Synthesis	Place and route	Timing analyser
CLA adder	23.18	19.00	23.44
CLA adder with place & route constraints	23.18	17.13	21.50

Table 5.4 CLA adder delay information

Module	Synthesis	Place and route	Timing analyser
Multiplier with place & route constraints	42.74	34.95	43.48

Table 5.5 Multiplier delay information

Module	Synthesis	Place and route	Timing analyser
Shifter	13.58	12.40	16.00
Shifter with place & route constraints	13.58	11.58	15.41

Table 5.6 Shifter delay information

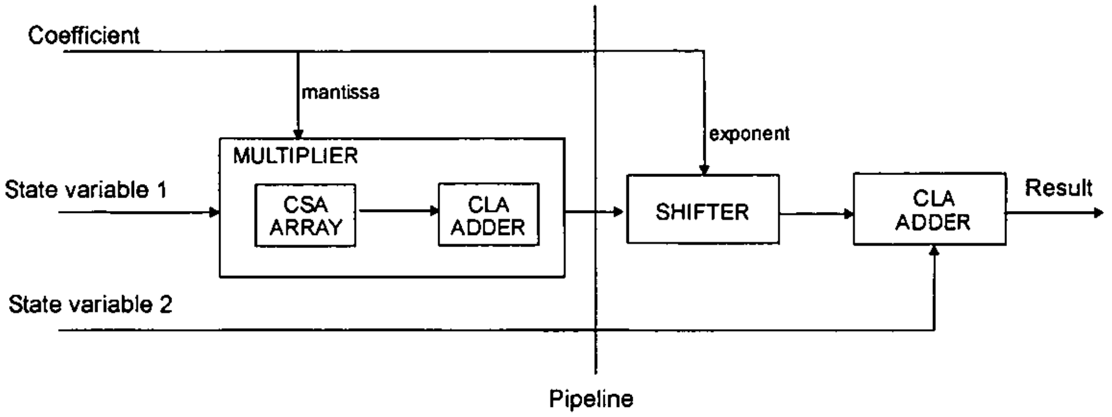


Figure 5.11 Pipelined MAC unit

Table 5.7 shows delay information for the complete MAC unit. The first row contains the delay of MAC unit without any optimisation. The second row indicates the delays of the pipelined MAC, and the third indicates the delays of the pipelined MAC with place & route constraints.

Module	Synthesis	Place and route	Timing analyser
MAC	75.61	61.64	76.41
Pipelined MAC	42.74	42.01	51.57
Pipelined MAC with place & route constraints	42.74	32.6	39.12

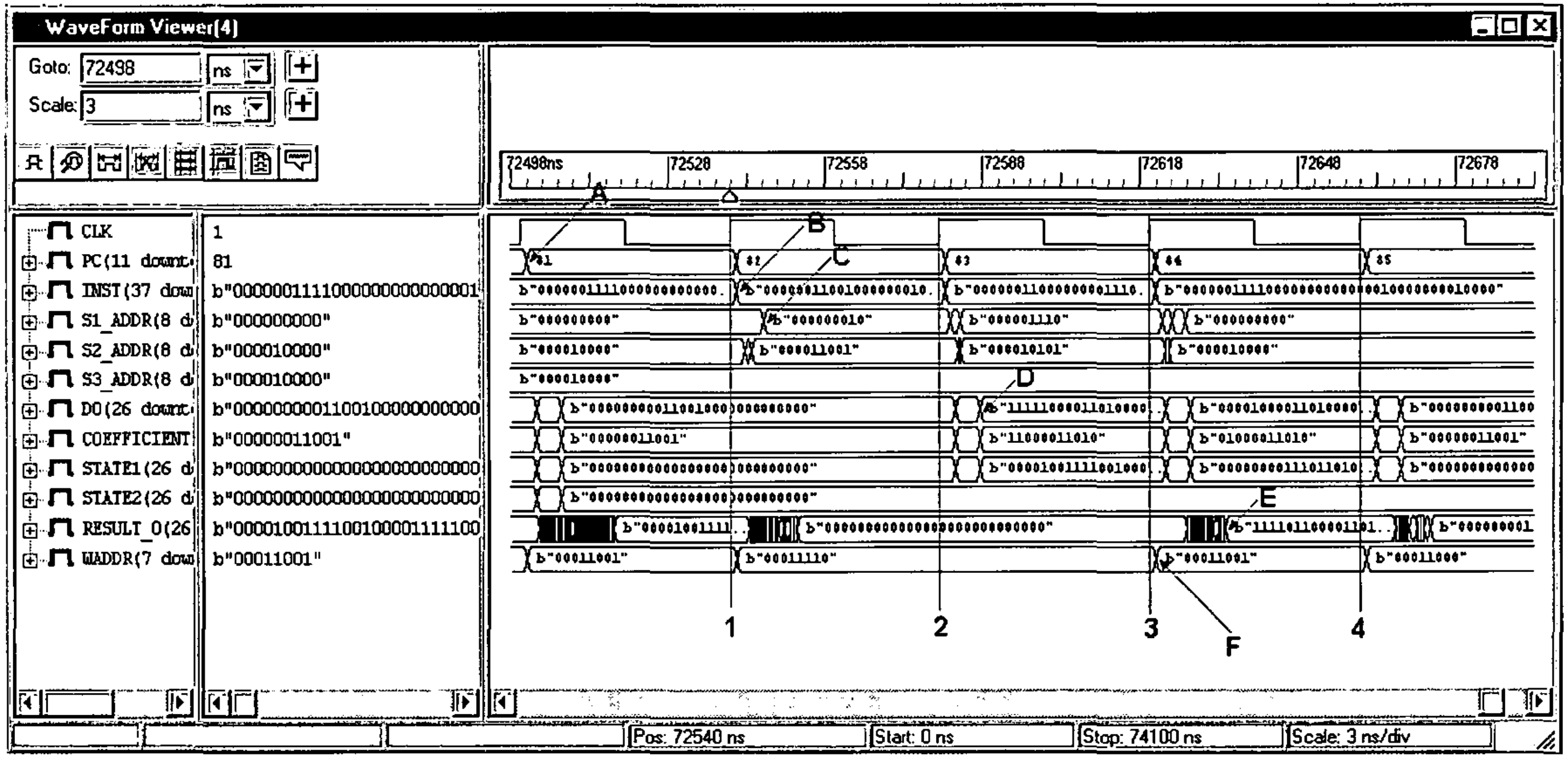
Table 5.7 MAC unit delay information

Because the MAC unit contains one pipeline stage, the MAC unit can process two sets of operands simultaneously. At the time that the operands specified by one instruction are being read from the data memory and transferred to the MAC unit inputs, the result from the previous instruction is produced at the output of the MAC unit and copied back to the data memory.

Figure 5.12 shows a simulation waveform of a MAC instruction cycle performed by the CSP. The program counter value (A) indicates the instruction to be executed. At rising edge 1, the instruction is fetched form the program memory (B) and the addresses it contains (C) are used to read data from the data memory at rising edge 2. The data (D) is then transferred to the MAC unit that produces a result after one clock cycle (E). The result of the operation is then stored in the data memory on rising edge 4, the location is indicated by the destination address (F) included in the MAC instruction.

Figure 5.13 shows a simulation waveform of the pipelined MAC unit. The MAC operation is performed in two clock cycles. The input data that was read from the data memory at the rising edge 1 is passed to the MAC unit (D). The coefficient is then divided into its mantissa and exponent parts (G). The state variable 1 and the mantissa used to feed the array of CSA adders to produce sum and carry vectors (H) that are added to complete the multiplication (I). The result of the multiplication is then stored in a pipeline register. After rising edge 2, the result of the multiplication is shifted according to the value of the exponent (J). And finally, the output of the shifter (K) is added to the state variable 2 (L) to obtain the result of the MAC operation (H). Note that to ensure that the correct result is obtained on the MAC operation, the exponent (J) and the state variable 2 (L) are also pipelined so their values are available on the second clock cycle.

Figure 5.14 shows the layout of the MAC unit on the ProASIC device and Figure 5.15 shows the complete CSP layout.



WaveForm Viewer(4)

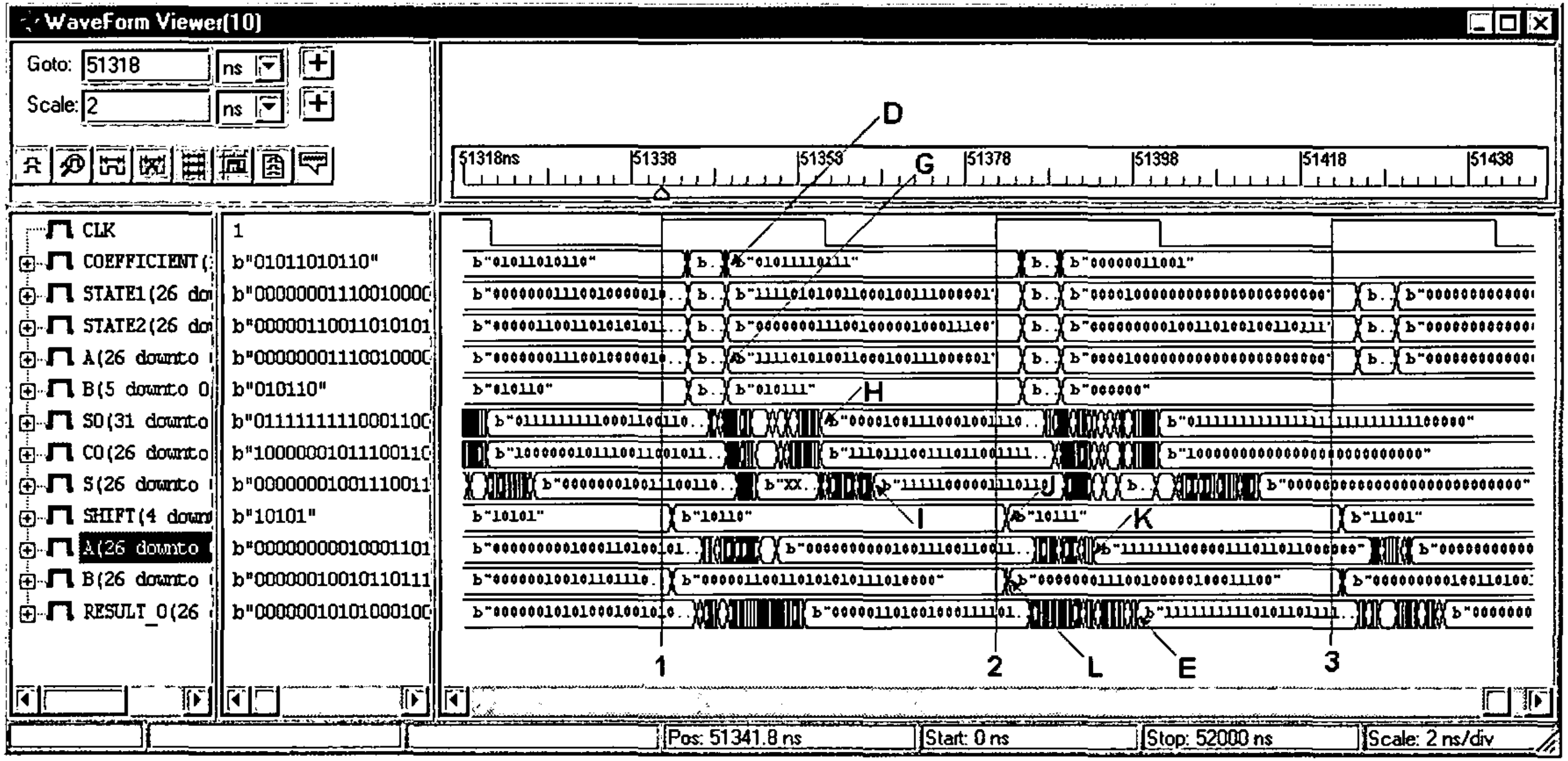
Goto: 72498 ns

Scale: 3 ns

CLK	1
PC(11 downt	81
INST(37 dow	b"0000001111000000000000001
S1_ADDR(8 d	b"000000000"
S2_ADDR(8 d	b"000010000"
S3_ADDR(8 d	b"000010000"
D0(26 downt	b"00000000011001000000000000
COEFFICIENT	b"00000011001"
STATE1(26 d	b"00000000000000000000000000
STATE2(26 d	b"00000000000000000000000000
RESULT_0(26	b"00001001111001000011111100
WADDR(7 dow	b"00011001"

Pos: 72540 ns | Start: 0 ns | Stop: 74100 ns | Scale: 3 ns/div

Figure 5.12 CSP simulation waveform



Goto: 51318 ns

Scale: 2 ns

CLK	1
COEFFICIENT(:	b"01011010110"
STATE1(26 do	b"00000001110010000
STATE2(26 do	b"00000110011010101
A(26 downto	b"00000001110010000
B(5 downto 0	b"010110"
S0(31 downto	b"01111111110001100
C0(26 downto	b"10000001011100110
S(26 downto	b"00000001001110011
SHIFT(4 down	b"10101"
A(26 downto	b"00000000010001101
B(26 downto	b"00000100101101111
RESULT_0(26	b"00000010101000100

Pos: 51341.8 ns | Start: 0 ns | Stop: 52000 ns | Scale: 2 ns/div

Figure 5.13 Pipelined MAC unit simulation waveform

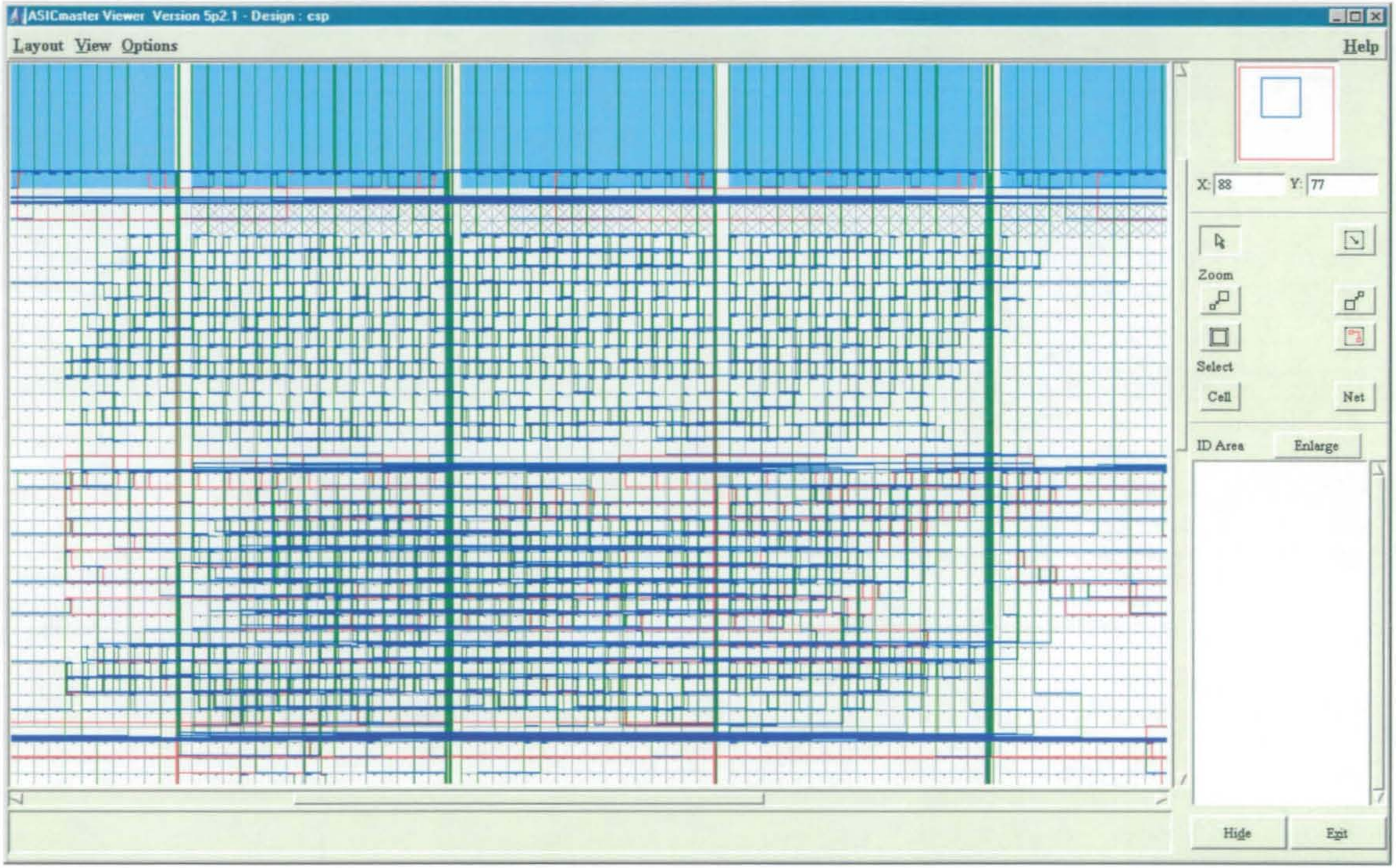


Figure 5.14 MAC unit layout

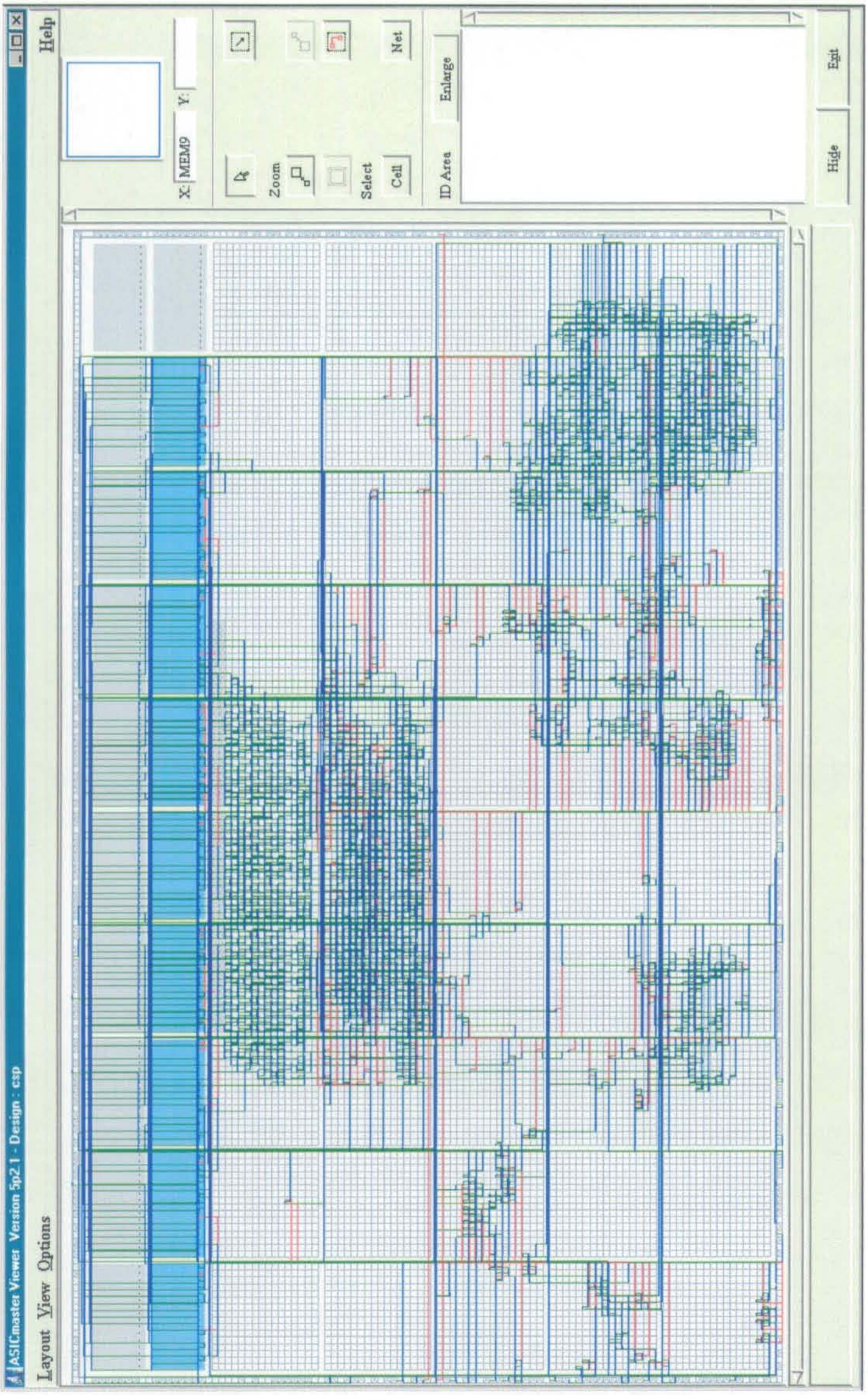


Figure 5.15 CSP layout

5.8.3 Hardware testing

The design was downloaded into an A500K130 device and verified for speed using a parallel tester. The parallel tester uses the test vectors to drive the design and compares the outputs against the expected output vectors provided by the user. The clock frequency and the strobe time are varied to produce a two-dimension plot (Shmoo Plot). The plot shows how the test passes or fails when both parameters are varied and the test is executed repeatedly.

According to the results obtained with the software tools, the delay of the critical path of the CSP under worst conditions is approximately 43ns (see Table 5.5), which corresponds to a maximum frequency of 25 MHz. However, the results obtained from the parallel tester were much better than expected (see Figure 5.16). In this figure, the passes are indicated by a '*' in the plot. The line formed with 'r' at the bottom of the graph indicates the minimum cycle time the parallel tester can operate at (20 ns), and the X and Y axis are shown as dotted lines. The maximum frequency in which the CSP operated correctly is 50 MHz, which is in fact, the maximum frequency the tester can operate at. This situation can be explained by the fact that at the time when the test were realised, the ProASIC devices were in the final stages of development prior to their market introduction.

```
                >                                  <
          50ns  +.....******************************+
                |.    ******************************|
                |.    ******************************|
                |.    ******************************|
                |.    ******************************|
          40ns  +.....******************************+
                |.    ******************************|
                |.    ****************************  .|
                |.    *************************     .|
Cycletime       |.    ***********************       .|
          30ns  +.....*********************.........+
                |.    ******************************|
                |.    **************** *************|
                |.    **************   *************|
                |.    ************     *************|
          20ns  +rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr+
                 +----------+-----------+----------+
                20ns       30ns        40ns       50ns
                              Strobe
```

Figure 5.16 Parallel tester results for the CSP (Shmoo Plot)

Despite the fact that the results provided by the software tools do not match the results obtained by the hardware tests. The main purpose of testing the design, which was to demonstrate that the design works properly at high frequencies, was accomplished.

5.9 SYSTEM INTERFACE

The processor will be embedded within the complete control system and will normally be programmed in a separate programming system (see Section 3.4.3). A group of analogue-digital and digital-analogue converters provide the interface to the physical system. A number of configuration options exist and the exact configuration will depend on specific system requirements and available resources.

Figure 5.17 shows just a proposed configuration where external data buses are used to connect the ADCs and DACs to the CSP input and output ports. This configuration allows sampling of analogue input at the same time, thus a single CSP control signal is required. Assuming that the outputs of the ADCs are registered, each input can be accessed at any time between consecutive samples because they are each stored in their own dedicated register. Another advantage of this configuration is that the number of I/O pins remains constant regardless of the number of input and output signals of the system.

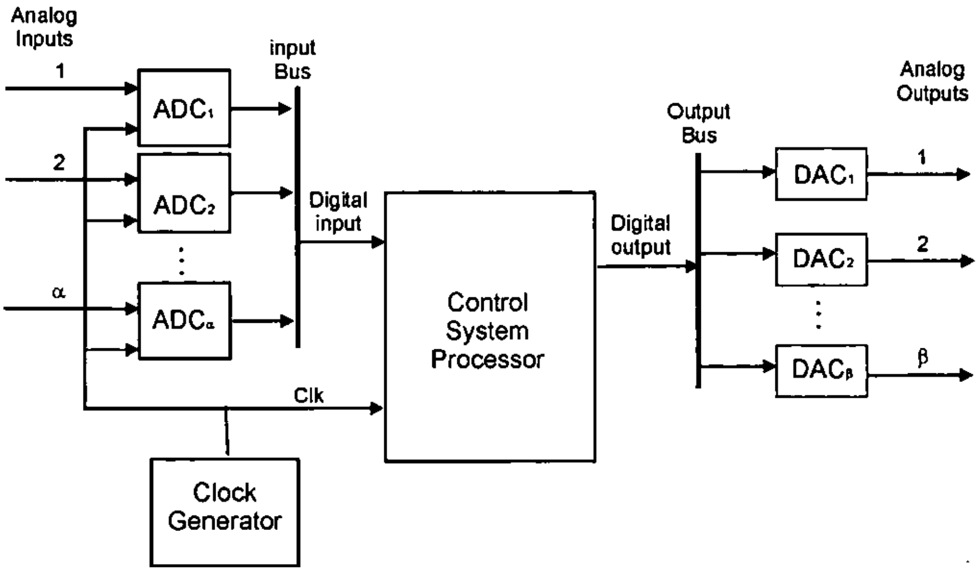


Figure 5.17 CSP interface

5.10 Summary of the Chapter

This chapter has described the proposed architecture to implement the CSP. It described the process used to create the architecture. Special attention was given to the design of the MAC unit, which performs high-speed multiply-and-accumulate operations on operands represented using the special formats described in Section 4.5. An analysis of the requirements and detailed low level design resulted in a compact MAC unit capable of producing one result per clock cycle. The memory system of the CSP consists of three separate memories, program memory, data memory and initial data memory. In order to provide data at the speed required by the MAC unit, the data memory was designed to support four data operations simultaneously (three read, one write).

The results of synthesising the design were presented. These results show that CSP can easily fit in medium range ProASIC device, which allows the possibility of integrating extra functions in required by a specific application. The CSP design was downloaded into an A500K130 ProASIC device and tested for speed. It was capable of running at 50MHz, which is about twice as fast as the speed estimated by the software tools.

In summary, the CSP has been designed to execute a well-defined control task, which is defined by the controller formulation explained in Chapter 4. This architecture takes advantage of our analysis of the specific control application to permit efficient and cost effective realisations of the required processing functions.

Chapter 6

CSP software

6.1 OBJECTIVES OF THE CHAPTER

In Chapter 4 we identified the operations that need to be executed to perform the control algorithm and in Chapter 5 a hardware architecture to perform those operations was developed. This chapter looks into the software that will implement the control algorithms within the CSP and the software environment needed to support this implementation. The objectives of this chapter are:

- To define the CSP instruction set
- To identify a suitable software structure for the CSP program
- To describe the supporting software suite

6.2 INSTRUCTION SET

6.2.1 Description of the instructions

Most of the operations needed to perform the control algorithm are the multiply-and-accumulate operations performed by the MAC unit described in Chapter 5; thus it is natural to have an instruction to perform such operation. The MAC instruction indicates to the CSP to perform the operation

$$R = (A*B) + C$$

where A is a coefficient and B, C and R are value represented in state variable format.

A READ instruction allows the CSP to read the initialisation values form data memory. It also reads input samples when the algorithm loop has begun. The values are copied into the register file location indicated by the instruction. A WRITE instruction is used to transfer a value from the register file to an output channel.

Finally, to provide support for unconditional jumps, the program counter unit requires an initial and a final address value. The WRITEPC instruction is used to copy those from the initialisation data memory into program counter registers. Table 6.1 summarises the CSP instruction set.

Mnemonic	Description
MAC	Multiply-and-accumulate operation
WRITEPC	Write to program counter registers
READ	Read input sample or initial values
WRITE	Extract output values

Table 6.1 CSP instruction set

The instruction formats used for the CSP instruction are very simple. The first field of all the instructions contains the operation code (OP) that indicates which instruction is to be executed. Table 6.2 shows the value of the 2-bit operation code for each instruction. A more detailed description of each instruction is given in the following sections.

Instruction	OP
MAC	00
WRITEPC	01
READ	10
WRITE	11

Table 6.2 Operation code for the CSP instructions

6.2.2 MAC instruction

The MAC instruction performs a multiply-and-accumulate operation. The addresses of the operands are included in the instruction fields. The second field contains the location in the register file where the result of the operation is to be placed. The following three fields indicate the location of the values involved in the operation (Figure 6.1).

OP	Destination	Source 1	Source 2	Source 3

Figure 6.1 MAC instruction format

Syntax: MAC R1, R2, R3, R4

Operation: R1 ← R2 * R3 + R4

Note that the order in which the operands are indicated in the MAC instructions determines which operands will be multiplied, and which will be added to the multiplication result. The *Source1* field must point to a coefficient value, and *Source2* and *Source3* fields must point to values in state variable format. The result of this operation will be stored in state variable format.

It was mentioned in Section 5.4 that the register file can be used to store some constants in order to add flexibility to the MAC unit operation. Some useful constant values are 0, 1 and -1. Note that to implement the control algorithm, it is not necessary to store these three constants in both formats (coefficient and state variable). However, they will be included in all the program and simulation examples to maintain consistency. The inclusion of these constants allows the MAC instruction to perform a number of different operations as shown in Table 6.3. Note that the CSP was not intended to support these operations. It is recognised that the MAC unit is not the best approach to implement these operations. However, the inclusion of a specialised unit to perform them operations is not justifiable because these operations are rarely needed.

Instruction	Syntax	Operation
No operation	MAC R2, 1, R2, 0	R2 ← R2
Move	MAC R2, 1, R3, 0	R2 ← R3
Addition	MAC R2, 1, R3, R4	R2 ← R3 + R4
Multiplication	MAC R2, R3, R4, 0	R2 ← R3 * R4
Sign invert	MAC R2, -1, R4, 0	R2 ← -1 * R4
Increment	MAC R2, 1, R4, 1	R2 ← R4 + 1
Decrement	MAC R2, 1, R4, -1	R2 ← R4 - 1

Table 6.3 Additional operations that can be implemented with the MAC instruction

6.2.3 READ instruction

The READ instruction contains four fields (Figure 6.2). The operation *OP* field identifies the instruction. The destination field indicates the section and location in the data memory where the input value is to be stored. The *Input Sel* field indicates where the input value is to be read from, data memory or input port. In the former case, the source field contains the location of the value in the memory. In the latter case, the value of the *Input Sel* field can be used to point to specific inputs in the case where several inputs channels are being used (see Section 5.9) and the *Source* field is not used.

OP	Destination	Input Sel	Source

Figure 6.2 READ instruction format

Syntax: READ R1, Sel, Saddr

Operation:

```
If Sel == 0
        R1 ← data[Saddr]          ; Read from data memory
else
        R1 ← ADC[Sel]             ; Read from ADC number Sel
```

6.2.4 WRITE instruction

The WRITE instruction contains only three fields (Figure 6.3): The *OP* field identifies the instruction, the *Output Sel* field that selects the appropriate output where the output value is transferred, and the *Source* field that contains the location of the output value in the state variable data memory. The information in the *Output Sel* field is only relevant when several output channels are being used.

OP	Output Sel	Source

Figure 6.3 WRITE instruction format

Syntax: WRITE Sel, R1

Operation: DAC[Sel] ← R1

6.2.5 WRITEPC instruction

The WRITEPC instruction also contains three fields (Figure 6.4). The *OP* field identifies the instruction, the *Destination* field that selects the appropriate register in the Program Counter (see Section 5.5.1) module where the input value is to be stored and the *Source* filed that contains the location of the input value in the data memory.

OP	Destination	Source

Figure 6.4 WRITEPC instruction format

Syntax: WRITEPC Sel, Addr

Operation:

```
if sel == 0
        pcstart ← data[Addr]        ; Update initial address register
else
        pcstop ← data[Addr]         ; Update final address register
```

6.3 SOFTWARE STRUCTURE

In this section the overall structure of the program that implements the control algorithm and the calculation schedule are described. The CSP program controls how the operations are sequenced to perform the algorithm. To generate the CSP program it is necessary consider the order in which operations must be done, the number of inputs, the number of outputs and the order of the control system to be implemented.

6.3.1 Program scheme

Although the CSP program is modified according to the system to be controlled, the overall scheme remains the same. The structure adopted to implement the digital controller program is shown in Figure 6.5. It is divided in two main parts: initialisation and algorithm loop. The program begins with an initialisation process where each controller state variable is set to zero, and the other variables used in the program are given initial values. All the coefficients and state variable initial values are transferred from the external data ROM to the register file. Also, the program counter registers that specify the initial and final instruction for the algorithm loop are updated. Finally, the program enters an infinite loop where the input samples are used to calculate the output values to the control system.

Due to the simplicity of the control algorithm, the CSP will not include support for subroutine calls. This means that all the instructions will be coded in the order of their execution within the main program loop. This approach avoids the delays associated with subroutine calls and will dramatically reduce the amount of hardware resources required to implement the processor.

The fact that the control algorithm to be implemented does not require data dependent branching operations facilitates the scheduling of the operations. A fixed schedule helps to achieve one of the main goals of this software scheme, which is to have a constant sample rate.

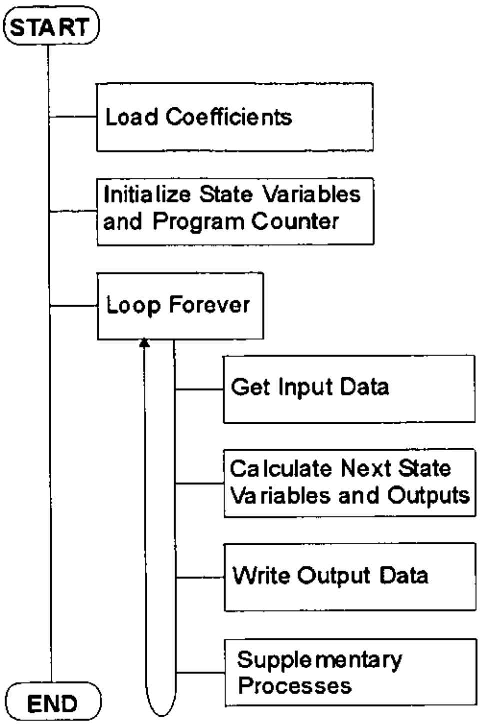


Figure 6.5 CSP program scheme

For a given clock frequency, the time between successive input samples or the time required to perform an entire loop of the program depends on the number of instructions executed and the number of clock cycles needed for each instruction. The main advantage of this structure is its simplicity, though any modification in the number of instruction within the loop will have an affect on the sampling period.

6.3.2 Calculation schedule

The overall sequence of operations within the algorithm loop, with the objective of minimising the computational delay between the arrival of the input **U** and generating the output **Y** is shown in Figure 6.6.

The calculation $\mathbf{CX}_{k+1}$ is performed once $\mathbf{X}_{k+1}$ is available and prior to the next sampling time. Thus after the new inputs are available, only $\mathbf{DU}_k$ needs to be calculated and added to the already calculated values of $\mathbf{CX}_{k+1}$ (now $\mathbf{CX}_k$) to obtain the output values $\mathbf{Y}_k$. The value of σ_{k+1} can be calculated by adding the state variables as soon as they are available. In this way after the next sample point (k → k+1) the value is already known.

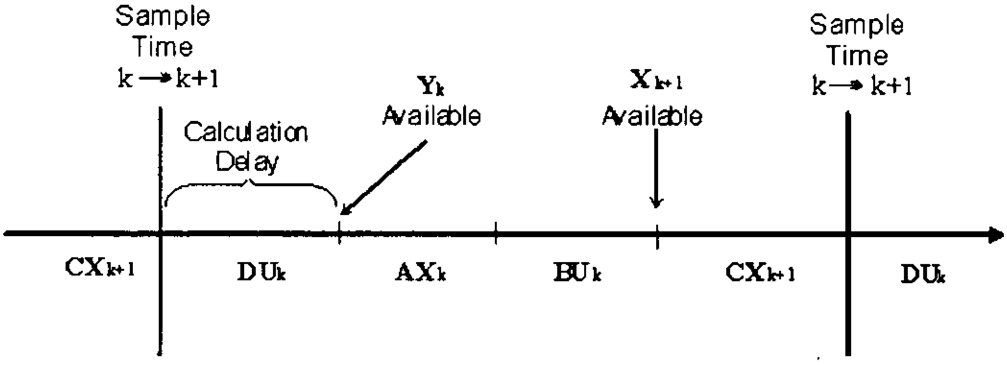


Figure 6.6 Calculation schedule within the algorithm loop

As an example consider a 2nd order filter. The state equations are:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & a_1 \\ -a_2 & 1-a_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} u \qquad (5.1)$$

$$y = \begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + d\,u \qquad (5.2)$$

The equations used to calculate the outputs and to update the state variables are (see Section 4.4.1):

$$y = c_1 x_1 + c_2 x_2 + du \qquad (5.3)$$

$$x_1 = x_1 + a_1 x_2 \qquad (5.4)$$

$$x_2 = x_2 - a_2(x_1 + x_2) + a_2 u \qquad (5.5)$$

When the input value is available, the output is obtained using Equation 5.3, which requires 3 MAC operations.

To follow the calculation schedule described in Figure 6.2, the previous equations are split into the following equations:

$$y = p + du \tag{5.6}$$

$$x_1 = x_1 + a_1 x_2 \tag{5.7}$$

$$x_2 = x_2 + a_2 \sigma + a_2 u \tag{5.8}$$

$$\sigma = -(x_1 + x_2) \tag{5.9}$$

$$p = c_1 x_1 + c_2 x_2 \tag{5.10}$$

Note that the product **CX** is calculated immediately after the state variables are updated and assigned to an auxiliary variable *p*. In the way, the new output value can be produced shortly after the next sampling time by adding *p* to the product **DU**. Note that after reading the input value, only one MAC instruction is now required to obtain the corresponding output value. The value of the auxiliary variable σ, which contains the accumulated value of the state variables, is obtained immediately after the state variables are updated (see Section 4.4.2).

The instructions required to implement these equations are shown in Figure 6.7.

```
READ  U, 0                  ; Read Input from ADC 1

MAC   Y, D, U, P            ; $y = p + du$

WRT   0, Y0                 ; Write Output to DAC 0

MAC   X1, A1, X2, X1        ; $x_1 = x_1 + a_1 x_2$
MAC   X2, A2, Rs, X2        ; $x_2 = x_2 + a_2 \sigma + a_2 u$
MAC   X2, A2, U, X2

MAC   Rs, 1, X0, X1         ; $\sigma = -(x_1 + x_2)$
MAC   Rs, -1, Rs, 0

MAC   P, C1, X1, 0          ; $p = c_1 x_1 + c_2 x_2$
MAC   P, C2, X2, P
```

Figure 6.7 Segment of a CSP program

A general example CSP program that can be used as a model to implement any controller using the selected formulation is shown in appendix A.

6.3.3 Data dependency

One problem generated by pipelining is data dependency, in which some sequences of instructions do not produce the expected results because the current operation requires a result form the previous operation which has not yet stored the result back into memory. To illustrate this, consider the following sequence of operations:

```
1. MAC R10, R1, R2, R3     ; R10 ← R1 * R2 + R3
2. MAC R11, R1, R2, R10    ; R11 ← R1 * R2 + R10
3. MAC R12, R4, R5, R6     ; R12 ← R4 * R5 + R6
```

Instruction 2 begins execution when the result of instruction 1 has not been written back. This means that instruction 2 will use the old value of R10 and therefore produce a wrong result. A rearrangement in the order of the instructions solves this problem, by executing instruction 3 before instruction 2, the result of instruction 1 is written back into the register file before it is read as a source operand by instruction 2. The new sequence of instruction is:

```
1. MAC R10, R1, R2, R3     ; R10 ← R1 * R2 + R3
2. MAC R12, R4, R5, R6     ; R12 ← R4 * R5 + R6
3. MAC R11, R1, R2, R10    ; R11 ← R1 * R2 + R10
```

Note that when the pipelined MAC is used in the CSP, an extra clock cycle is needed to complete an instruction (see Section 5.7), therefore it is necessary to insert an extra instruction after instruction 1 to ensure proper operation. This rearrangement of instructions, which is done manually, results in additional design effort creating a CSP program.

6.3.4 CSP program size

The number of instructions required to perform the control algorithm depends on the complexity of the controller. The sub-tasks contained within the CSP program are listed below indicating the number of CSP instructions required to perform each task.

Task	**Number of CSP instructions**
Load Coefficient values	$n + n\alpha + \beta n + \alpha\beta + 3$
Initialise state variables	$\alpha + 2\beta + n + 3$
Initialise PC values	2
: Algorithm cycle start	
Get Input Data	α
Calculate $\mathbf{DU}_N$	$\alpha\beta$
Calculate **Yn**	β
Write Output Data	β
Calculate $\mathbf{X}_{N+1}$	$(2 + \alpha)n$
Calculate $\mathbf{AX}_N$	
Calculate $\mathbf{BU}_N$	
Calculate $\mathbf{CX}_{N+1}$	$n\beta$
: End Algorithm cycle	

where α,β and n are defined as the number of input, the number of output and the number of internal state variables respectively. As an example consider a 4th order single-input single-output controller ($\alpha = 1, \beta = 1, n = 4$), the number of instructions required to perform the operations within the algorithm loop is 20. The number of instruction within the algorithm loop and the frequency the CSP operates at will determine the speed at which input samples are processed. For example, if the CSP requires 20 instructions per algorithm cycle and the processor runs at 20MHz, the maximum sampling frequency is 1MHz.

6.4 SOFTWARE SUITE

6.4.1 CSP model

The purpose of the CSP Model is to provide a clear understanding of the algorithm and its numerical requirements, as well as a verified functional specification of the processor and test vectors to verify the hardware design. Input data to the model is provided from Matlab simulations or from the CSP signal generator and the program to be simulated is generated by the CSP program generator.

6.4.2 Signal generator

The CSP signal generator provides input test data to the CSP model. One of the basic analysis and design requirements is to evaluate the response of a system for a given input. Test input signals are used, both analytically and during testing, to verify the design of a control system. It is not practical to choose complicated input signals to analyse performance. Thus, usually standard test inputs are used. These inputs are impulses, steps, ramps, parabolas and sinusoids [Nise00].

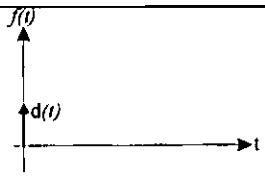
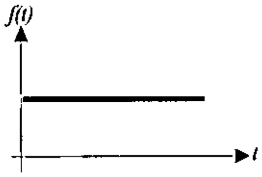
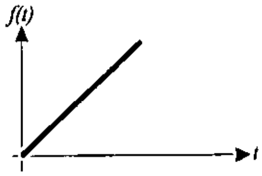
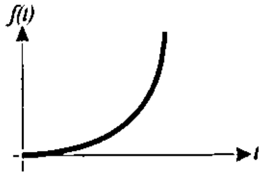
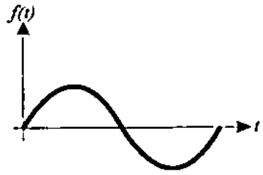
Input	Function	Sketch
Impulse	*d(t)*	
Step	*u(t)*	
Ramp	*tu(t)*	
Parabola	$\frac{1}{2}t^2u(t)$	
Sinusoid	Sin ωt	

Table 6.4 Test signal generated by the data generator

6.4.3 Program generator

The CSP program is created using the program generator. The number of instructions varies according to the number of inputs, outputs and order of the system. Thus, each calculation part is generated using these parameters to modify the source and destination addresses for each instruction. The program is generated in text format and then converted into VHDL (as a ROM element) and added to the VHDL code. This is then synthesised and placed and routed. Figure 6.8 shows a CSP that implements a 2nd order SISO controller like the one used as example in Section 4.4. The order in which the instructions are shown in this example program was chosen to facilitate the identification of the calculations required. However some instruction rearrangement is needed to avoid data dependency problems. A detailed sequence of instructions needed to perform the control algorithm is that illustrated in a template program shown in Appendix A.

```
READ     CF0   Pt0   DROM_C0              ; Copy coefficients
READ     CF1   Pt0   DROM_C1              ; to data memory
READ     CF_1  Pt0   DROM_C2
READ     A1    Pt0   DROM_C3
READ     A2    Pt0   DROM_C4
READ     B1    Pt0   DROM_C5
READ     B2    Pt0   DROM_C6
READ     C1    Pt0   DROM_C7
READ     C2    Pt0   DROM_C8
READ     D     Pt0   DROM_C9

READ     SV0   Pt0   DROM_S0              ; Initialise state
READ     SV1   Pt0   DROM_S1              ; variables
READ     SV_1  Pt0   DROM_S_1
READ     X1    Pt0   DROM_S0
READ     X2    Pt0   DROM_S0
READ     Y     Pt0   DROM_S0
READ     U     Pt0   DROM_S0
READ     Rs    Pt0   DROM_S0
READ     Acc   Pt0   DROM_S0

READ     tmp1  Pt0   DROM_P 0             ; Initialise program
READ     tmp2  Pt0   DROM_P 1             ; counter
WRITEPC  PC1   1     tmp1   0
WRITEPC  PC2   1     tmp2   0
                                          ; Algorithm loop
READ     U     IPt1                       ; Read Inputs
MAC      Y     D     U      P             ; Calculate DUk
WRITE    OPt0  Y                          ; Write Outputs
MAC      X2    A2    X1     X2            ; Calculate
MAC     X1   A1   Rs    X1        ; Xk+1 = AXk + BUk
MAC     X2   B2   U     X2
MAC     Acc  1    0     X2
MAC     X1   B1   U     X1
MAC     Rs   1    Acc   X1
MAC     Rs   -1   Rs    0         ; Calculate Rs = -Rs
MAC     P    C2   X2    0         ; Calculate CXk+1
MAC     P    C1   X1    P
                                  ; End algorithm loop
```

Figure 6.8 CSP program that implements a 2nd order SISO controller

6.5 Summary of the Chapter

This chapter looked into the program that implements the control algorithms within the CSP and the software environment used to support this implementation. It explained the software scheme adopted to implement the control algorithm and the CSP instruction set.

The sequence of operations performed within the algorithm loop allows minimising the computation delay between the arrival of the sample inputs and generating the corresponding outputs. The problem of data dependency was also explained together with a simple procedure to solve it.

The reduced instruction set and CSP architecture allows us to implement the control algorithm in a very simple way. The reduced number of operations required to implement the control algorithm results in a short CSP program, where the actual number of instructions is determined by the control system characteristics. The operations to be performed will be indicated by the program and realised recursively with the state variables updated for the next step and output produced during each algorithm loop.

Chapter 7

CSP system test and benchmark

7.1 OBJECTIVES OF THE CHAPTER

This chapter presents the results of benchmarking the CSP against other processors and some simulations results. The objectives are:

- To show that the CSP can satisfy a range of high sample rate controller examples
- To prove the numerical aspects of its operation
- To describe the controller examples used to evaluate the CSP
- To benchmark the CSP against other processors running the same algorithms

7.2 METHODOLOGY

A comprehensive set of tests has been undertaken to prove the CSP's operation for a variety of filter types over a range of input conditions. A Matlab program is used to implement and simulate some example controllers using 32-bit floating-point variables to represent the coefficients and state variables. Additionally, a hardware implementation of the CSP is tested using the same input signals to compare the results against those obtained with the Matlab program.

Additionally, the CSP performance is compared against the performance of some popular commercially available processors. The processors included in the benchmark are listed in Table 7.1. To evaluate the performance of these processors, the controller examples were programmed in C and compiled to produce assembly code targeted at each processor. The assembly code is then analysed to produce an estimate of the computation time for each example and compare the results against those obtained with the CSP.

Manufacturer	**Processor**	**Device type**
Texas Instruments	TMS320C31	Digital signal processor
Texas Instruments	TMS320C54	Digital signal processor
Infineon	C167	Microcontroller
Intel/ARM	Strong-ARM	General-purpose processor
Intel	PentiumIII	General-purpose processor

Table 7.1 Processors included in the benchmark

The benchmark includes a comparison of the computation time, the number of instruction required to perform the control algorithms, average clock cycles needed to perform an instruction. Additionally, a comparison table that includes data such as power consumption, voltage supply, technology and hardware complexity, is presented.

7.3 EXAMPLE CONTROLLERS DESCRIPTION

7.3.1 Validation example - a 4th order 1Hz Butterworth low pass filter

A general-purpose single-input single-output filter has been as chosen an example to assess the CSP's performance. This example was also used to explore the limits of the numerical formulation provided within the CSP. Sample frequencies of 100Hz, 1kHz, 5kHz and 10kHz were used for testing, with results for some frequencies presented here. The transfer function of the filter is:

$$H(s) = \frac{1}{\left(1 + 1.4\frac{s}{w} + \frac{s^2}{w^2}\right)^2} \tag{7.1}$$

where $w = 2\pi$. The transfer function was converted into the modified δ form. Figure 7.1 shows a diagrammatic representation of the filter. Appendix B gives the sets of coefficients for the sample frequencies used in the simulations.

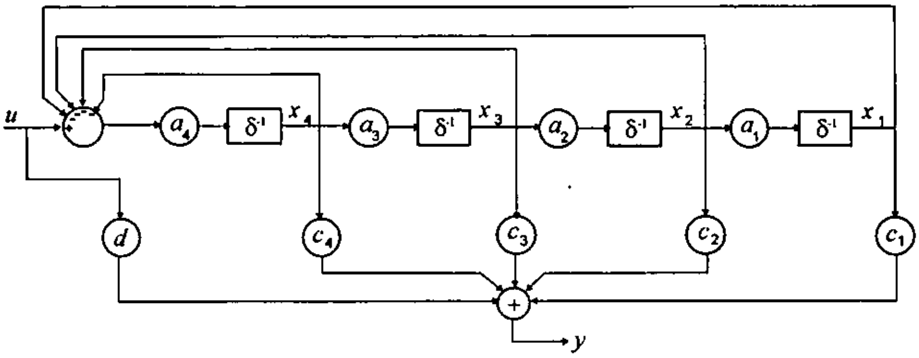


Figure 7.1 4th order SISO filter in modified δ form

7.3.2 Example controllers

Two medium and one high order example controller have been selected to establish the performance of the CSP, drawn from real control applications. The purpose is to demonstrate that the CSP can implement multiple-input multiple-output controllers satisfactorily. The performance results of these implementations are included in the benchmark.

7th order two-input two-output controller

This controller resulted from a H_∞ design to provide robust control performance for an industrial process control application, and although some simulation tests have been undertaken it is included principally for the purposes of benchmarking. It is a typical example of the kind of controller generated by modern control system design methods, with interaction between both inputs and the outputs, and of higher dynamic complexity than normally generated by classical control approaches. It has a number of closely related eigenvalues between 0.1 and 1Hz, and when operating at sample frequencies of 1kHz and higher represents a difficult control processing requirement. Figure 7.2 shows a diagram of the controller.

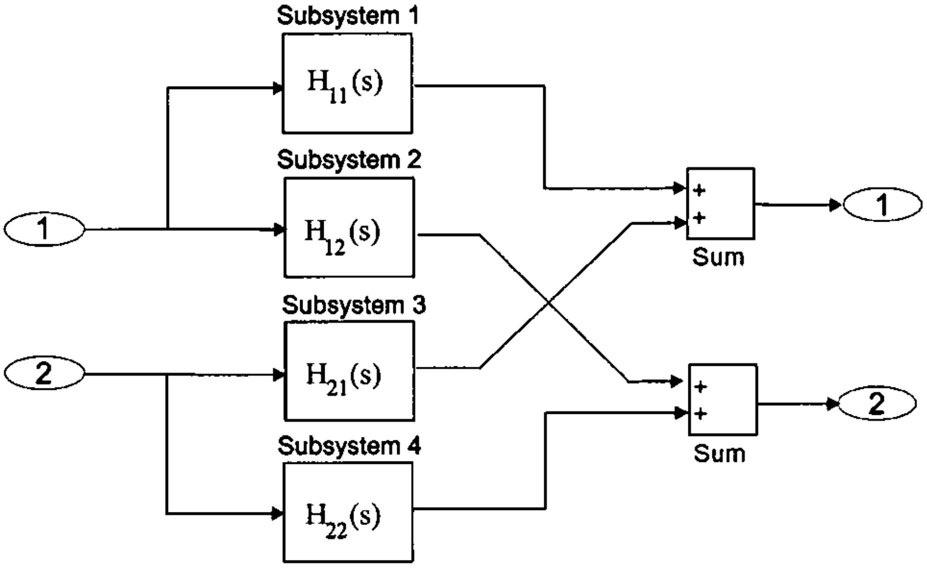


Figure 7.2 7th order two-input two-output example controller

The corresponding transfer functions are:

Subsystem 1:

$$H_{11}(s)=\frac{-0.0013s^7-0.0258s^6-0.2065s^5-0.9244s^4-2.4146s^3-3.6751s^2-2.7490s-0.7462}{0.0001s^7+0.0024s^6+0.0237s^5+0.1385s^4+0.5086s^3+1.1917s^2+1.6608s+1.1218}$$

Subsystem 2:

$$H_{12}(s)=\frac{-0.0005s^7-0.0094s^6-0.0726s^5-0.3177s^4-0.8484s^3-1.3781s^2-1.2358s-0.4522}{0.0001s^7+0.0024s^6+0.0237s^5+0.1385s^4+0.5086s^3+1.1917s^2+1.6608s+1.1218}$$

Subsystem 3:

$$H_{21}(s)=\frac{0.0007s^7+0.0130s^6+0.0996s^5+0.4158s^4+0.9593s^3+1.1684s^2+0.4179s-0.1197}{0.0001s^7+0.0024s^6+0.0237s^5+0.1385s^4+0.5086s^3+1.1917s^2+1.6608s+1.1218}$$

Subsystem 4:

$$H_{22}(s)=\frac{-0.0004s^7-0.0066s^6-0.0583s^5-0.3110s^4-1.0679s^3-2.3202s^2-2.9721s-1.7233}{0.0001s^7+0.0024s^6+0.0237s^5+0.1385s^4+0.5086s^3+1.1917s^2+1.6608s+1.1218}$$

Appendix B gives the sets of coefficients for the sample frequencies used in the simulations.

13th order three-input one-output Maglev loop controller

This example is a classically-designed active suspension controller having a single output and three inputs, i.e. a main feedback signal plus two additional inner feedback signals in a cascade feedback structure. The classical design approach means that the controller is in block structured form and the various transfer function elements in the controller have frequencies that vary from 0.1Hz to 20Hz. Figure 7.3 shows a diagram of the controller.

46th order twelve-input four-output Maglev vehicle Controller

The most complex example tested is a classically designed controller. It provides the control of the vertical modes of a magnetically suspended vehicle [Goodall78]. This was selected because it is a real example originally implemented in analogue form. It was the most dynamically-complex control example which could be found, and having a multi-input multi-output formulation provided a very demanding example to benchmark the CSP against the other processors. Figure 7.4 shows a diagram of the controller.

The value of the parameters indicated in Figure 7.3 and 7.4 and the set of coefficients used to implement these controller examples can be found in Appendix B.

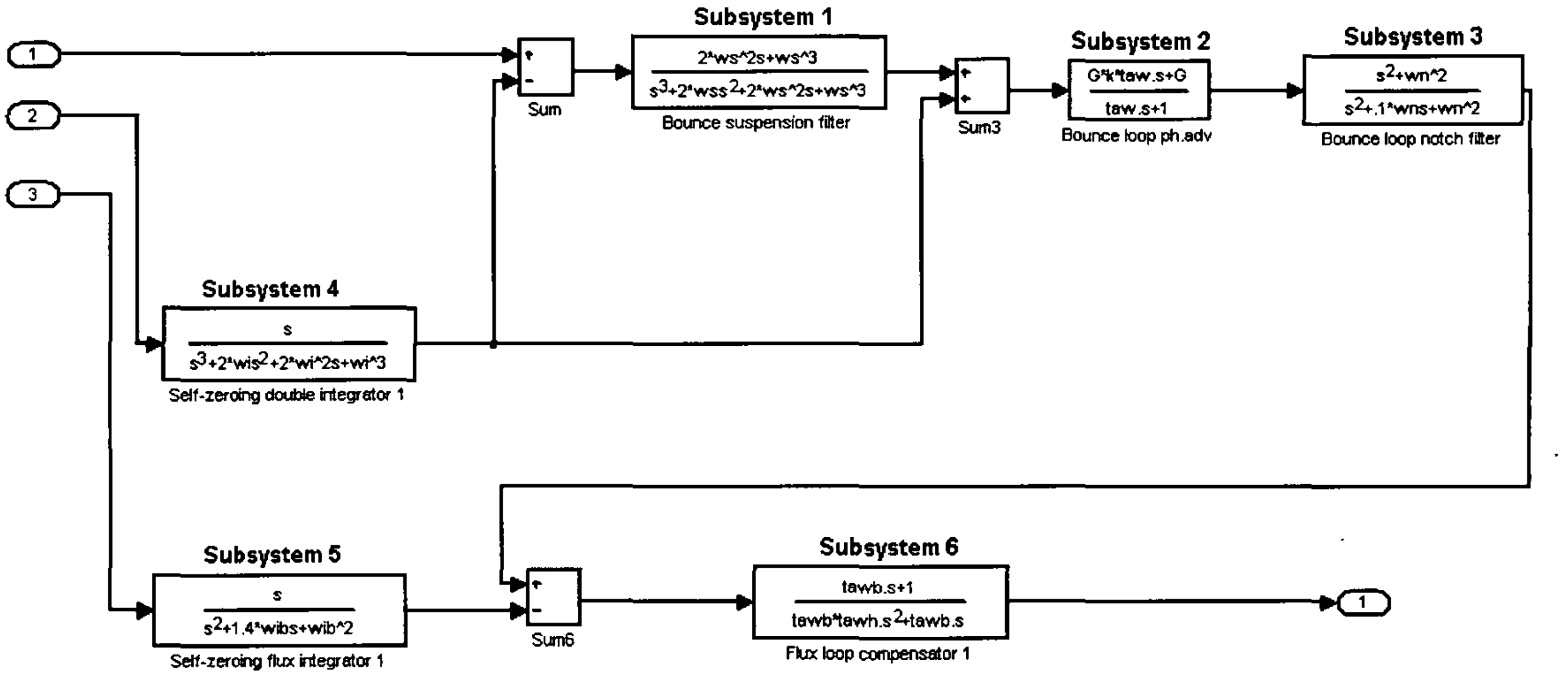


Figure 7.3 13th order three-input one-output Maglev loop controller

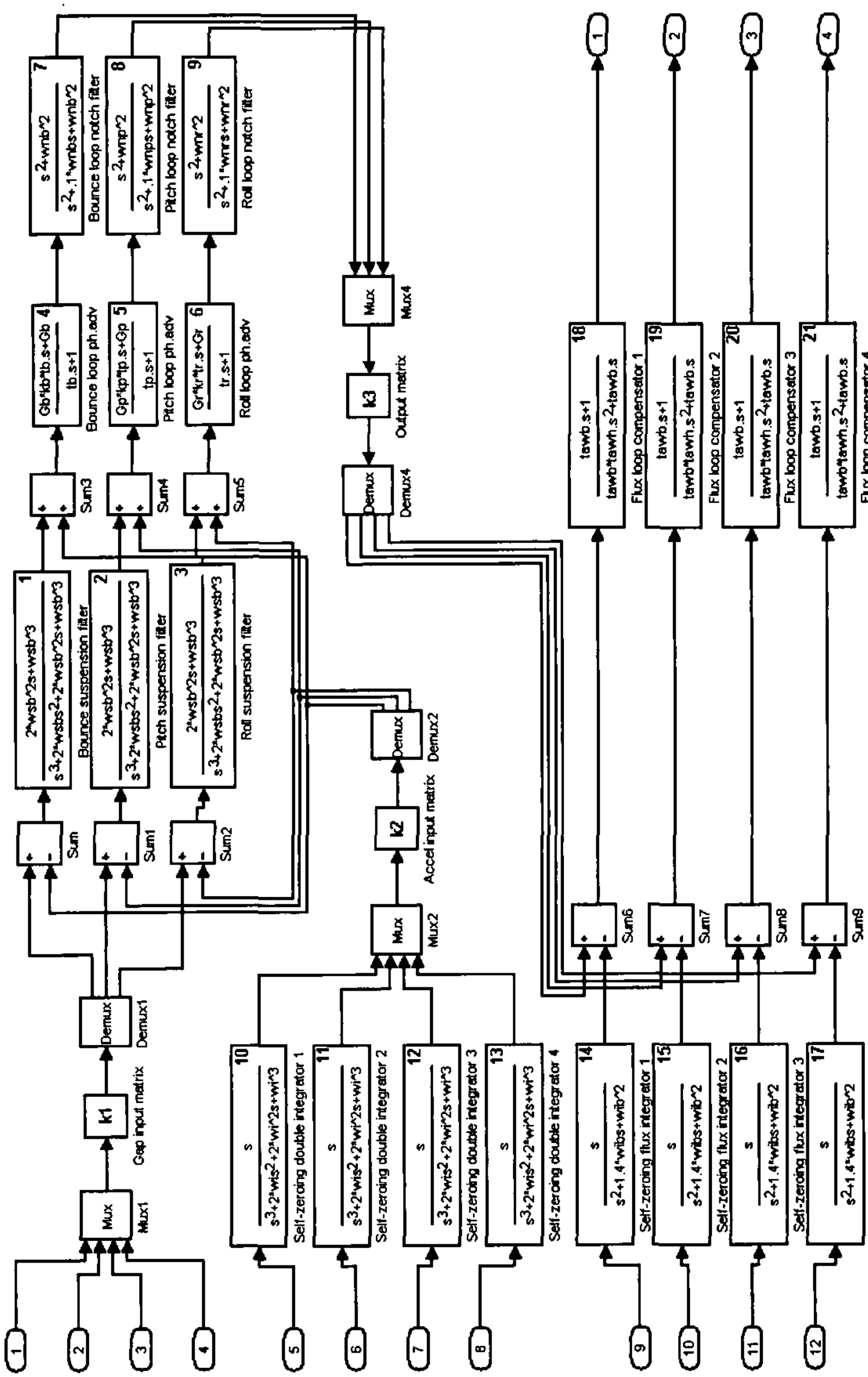

1
2
3
4
Mux
Mux1
k1
Gap input matrix
Demux
Demux1
Sum
Sum1
Sum2
2*wsb^2s+wsb^3 / s3+2*wsbs2+2*wsb^2s+wsb^3 1
Bounce suspension filter
2*wsb^2s+wsb^3 / s3+2*wsbs2+2*wsb^2s+wsb^3 2
Pitch suspension filter
2*wsb^2s+wsb^3 / s3+2*wsbs2+2*wsb^2s+wsb^3 3
Roll suspension filter
Sum3
Sum4
Sum5
Gb*kb*tb.s+Gb / tb.s+1 4
Bounce loop ph.adv
Gp*kp*tp.s+Gp / tp.s+1 5
Pitch loop ph.adv
Gr*kr*tr.s+Gr / tr.s+1 6
Roll loop ph.adv
s2+wnb^2 / s2+.1*wnbs+wnb^2 7
Bounce loop notch filter
s2+wnp^2 / s2+.1*wnps+wnp^2 8
Pitch loop notch filter
s2+wnr^2 / s2+.1*wnrs+wnr^2 9
Roll loop notch filter
5
6
7
8
s / s3+2*wis2+2*wi^2s+wi^3 10
Self-zeroing double integrator 1
s / s3+2*wis2+2*wi^2s+wi^3 11
Self-zeroing double integrator 2
s / s3+2*wis2+2*wi^2s+wi^3 12
Self-zeroing double integrator 3
s / s3+2*wis2+2*wi^2s+wi^3 13
Self-zeroing double integrator 4
Mux
Mux2
k2
Accel input matrix
Demux
Demux2
Mux
Mux4
k3
Output matrix
Demux
Demux4
9
10
11
12
s / s2+1.4*wibs+wib^2 14
Self-zeroing flux integrator 1
s / s2+1.4*wibs+wib^2 15
Self-zeroing flux integrator 2
s / s2+1.4*wibs+wib^2 16
Self-zeroing flux integrator 3
s / s2+1.4*wibs+wib^2 17
Self-zeroing flux integrator 4
Sum6
Sum7
Sum8
Sum9
tawb.s+1 / tawb*tawh.s2+tawb.s 18
Flux loop compensator 1
tawb.s+1 / tawb*tawh.s2+tawb.s 19
Flux loop compensator 2
tawb.s+1 / tawb*tawh.s2+tawb.s 20
Flux loop compensator 3
tawb.s+1 / tawb*tawh.s2+tawb.s 21
Flux loop compensator 4
1
2
3
4

Figure 7.4 46th order twelve-input four-output Maglev Vehicle Controller

7.4 REVIEW OF SELECTED PROCESSORS

7.4.1 Texas Instruments' TMS320C31

The Texas instruments' TMS320C31 is 32-bit floating-point digital signal processors. It is targeted at digital audio, data communications, and industrial automation and control. The data path consists of a multiplier, a barrel shifter and ALU. Its has a large address space, multiprocessor interface, one external interface port, two timers, one serial port, and multiple-interrupt structure. It can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. It also possesses a general-purpose register file, a program cache, internal dual-access memories, one DMA channel supporting concurrent I/O, and a short machine-cycle time [Texas00a].

7.4.2 Texas instruments' TMS320C54

The Texas instruments' TMS320C54 is a 16-bit fixed-point digital signal processor. It is designed to support personal and portable products like digital music players, 3G cell phones, and digital cameras as well as MIPS-intensive voice and data applications and single-channel applications. It has a modified Harvard architecture that has one program memory bus and three data memory buses. It also provides an ALU that has a high degree of parallelism, application-specific hardware logic, on-chip memory, on-chip peripherals, and RISC-like instruction set [Texas00b].

7.4.3 Infineon's C167

The Infineon's C167 is a 16-bit fixed-point microcontroller. It is one of the world's most successful 16-bit architectures. It is targeted towards low cost applications and is found in real-time embedded control applications such as automotive, industrial control, computer peripherals and data communications. Its main key features are:

RISC register based architecture, 16-bit CPU with 4 stage pipeline and jump cache, 32 bit bus to internal ROM, and von Neumann address space [Infineon00].

7.4.4 Intel's StrongARM SA-110

The Intel's StrongARM SA-110 processor is a 32-bit microprocessor targeted towards low power applications. It is used in a wide range of embedded applications, including high-bandwidth network switching, intelligent office machines, storage systems, remote access devices, Internet appliances, smart handheld, handheld personal computers, and mobile phones [Intel00a].

7.4.5 Intel's Pentium III

The Intel's Pentium III processor is a 32-bit floating-point processor. The Pentium is targeted at general-purpose desktop and mobile computing. It has a superscalar architecture, large on-chip caches, 64-bit data bus, extended instruction set that includes instructions optimised for signal processing, and branch prediction logic. The Pentium has been described as having a RISC core for a subset of its instructions, but in reality the Pentium contains a mixture of hard-wired simple instructions and microcoded complex instructions [Intel00b].

Processor	Data format	Frequency (MHz)	Main applications
CSP	Special format	50	Real-time control
TMS320C31	32-bit floating point	60	Digital audio, data communications, and industrial automation and control
TMS320C54	16-bit fixed-pint	160	Portable products, voice and data applications
C167	16-bit fixed-pint	25	Automotive, computer peripherals, industrial control and data communications
Strong-ARM	32-bit fixed-point	233	Embedded applications
Pentium III	32-bit floating point	500	Desktop and mobile computing

Table 7.2 Summary of processors' features

7.5 BENCHMARKING

7.5.1 Assumptions and considerations for benchmarking

The real-time code for these processors has been carefully assessed to ensure that a comparison as fair as possible is presented. Figure 7.5 shows the main routine of a C program that implements the 4th order filter described in Section 7.3.1.

```
void main()
{
        do
        {
                U = read_input();

                Y =  P + D*U;

                X[3]  = X[3]  + A[3]*X[2]  + B[3]*U;
                X[2]  = X[2]  + A[2]*X[1]  + B[2]*U;
                X[1]  = X[1]  + A[1]*X[0]  + B[1]*U;
                X[0]  = X[0]  - A[0]*Rs    + B[0]*U;
                Rs = X[3]  + X[2]  + X[1]  + X[0];
                P = C[0]*X[0]  + C[1]*X[1]  + C[2]*X[2]  + C[3]*X[3];

                write_output(Y);

        } while(TRUE)
}
```

Figure 7.5 C program for a 4th order SISO filter

Table 7.3 shows the resolution and data format used to represent the coefficients and state variables for each processor. For the CSP, 11-bit coefficient (6 bits for the mantissa and 5 bits for the exponent) and a 27-bit state variable format were used (see Chapter 4). The table also shows the resolution used to perform the multiplication of a coefficient and a state variable. As mentioned in Section 4.5 it is possible to find coefficient and state variable word lengths that can be represented with the data types provided by the processor, which satisfies the system response requirements. Thus, instead of emulating the special word formats used to represent the data within the CSP, the C programs use data types supported by the processors to perform the calculations.

Processor	State Variable	Coefficient	Multiplication
CSP	27-bit fixed-point	11-bit floating point	27x5 mixed format
TMS320C31	32-bit float	32-bit float	32 x 32 floating-point
TMS320C54	32-bit integer	16-bit integer	32 x 16 fixed-point
C167	32-bit integer	16-bit integer	32 x 16 fixed-point
Strong-ARM	32-bit integer	32-bit integer	32 x 32 fixed-point
Pentium III	32-bit float	32-bit float	32 x 32 floating-point

Table 7.3 Data format used to represent the state variables and coefficients for each processor

All the inputs are specified as signed integers within the 12-bit input/output variable range. The output values produced by the CSP, which are extracted from an internal variable in state variable format, have been rounded to the nearest integer. This means that if the value of the underflow bits is greater than or equal to 0.5, the output value will be increased by 1 (see Section 5.4.2)

The same general structure of the C program shown in Figure 7.5 was used to program the higher order controllers. Each program was compiled and optimised to produce assembly code so the number and type of instructions required to perform the algorithm can be identified. The compilers used to produce the assembly code for each processor are shown in Table 7.4.

Processor	Compiler	Manufacturer
TMS320C31	Code composer studio	Texas Instruments
TMS320C54	Code composer studio	Texas Instruments
C167	Keil C compiler	Keil Software
Strong-ARM	High C/C++ compiler for ARM	Metaware
Pentium III	Developer studio, Visual C++	Microsoft

Table 7.4 Compilers used to generate the assembly code used to evaluate the processors' performance

Modern processors include features that allow them to input and output continuous streams of data efficiently. Thus, it is assumed that the processes of reading the sampled inputs and writing the output values from/to the I/O interface require a single load/store instruction, which can be performed in a single instruction cycle. This assumption may not be strictly true for some processors. However, the computation time is mostly determined by the number of instructions required to complete algorithm loop that is much higher than the number of I/O instructions.

7.5.2 Benchmark results

This section summarises the results of the benchmark. Tables 7.5 to 7.8 show the computation times and maximum frequencies that the CSP and the other processors can achieve for each of the filter and controller examples described in section 7.3. The first column of these tables presents the processors included in the benchmark. The second column indicates the average clock cycles in which the processors perform an instruction. The average was obtained by dividing the total number of clock cycles required to perform a program loop by the number of instructions within the loop. The third and fourth columns indicate the number of instructions and time required by the processor to complete an algorithm cycle. Finally, the last column indicates the maximum sample rate that can be achieved by the processors. The computation time is obtained by multiplying the following three values: clock period, clock cycles per instruction and number of instructions.

The sample frequencies obtained with the CSP are 2MHz for the 4th order filter, and even for the complex multi-input multi-output 46th order controller a remarkably high sample frequency of 170kHz is possible.

Processor	Average clock cycles per instruction	Number of instructions	Computation time (μs)	Maximum sample frequency (kS/s)
CSP-50	1	23	0.460	2173
TMS320C31	2	48	1.603	623
TMS320C54	1.49	450	4.190	238
C167	3.34	194	25.920	38.5
Strong-ARM	1.79	43	0.331	3021
Pentium III	1.15	49	0.113	8823

Table 7.5 Benchmark results for the 4th order filter

Processor	Average clock cycles per instruction	Number of instructions	Computation time (μs)	Maximum sample frequency (kS/s)
CSP-50	1	54	1.080	925
TMS320C31	2	134	4.475	223
TMS320C54	1.49	882	8.213	121
C167	3.17	529	67.077	14.9
Strong-ARM	1.99	118	1.007	992
Pentium III	1.16	134	0.310	3216

Table 7.6 Benchmark results for the 7th order controller

Processor	Average clock cycles per instruction	Number of instructions	Computation time (μs)	Maximum sample frequency (kS/s)
CSP-50	1	75	1.500	666
TMS320C31	2	183	6.112	163
TMS320C54	1.49	1058	9.893	101
C167	3.47	823	114.240	8.75
Strong-ARM	1.69	160	1.162	860
Pentium III	1.19	191	0.456	2193

Table 7.7 Benchmark results for the 13th order controller

Processor	Average clock cycles per instruction	Number of instructions	Computation time (μs)	Maximum sample frequency (kS/s)
CSP-50	1	293	5.860	170.6
TMS320C31	2	672	22.444	44.5
TMS320C54	1.49	3745	35.000	28.5
C167	3.12	2890	361.600	2.76
Strong-ARM	1.72	598	4.418	226.3
Pentium III	1.09	779	1.700	588

Table 7.8 Benchmark results for the 46th order controller

Table 7.6 summarises how the CSP compares with the other processors. The computation time shown in tables 7.5 to 7.8 have been normalised to that of the CSP running at 50MHz. The closest DSP in performance is the TMS320C31 device, which still takes 3.48 times as long to compute the 4th order filter example, while the TMS320C54 fixed-point DSP takes 9.1 times as long as the CSP. The C167 microcontroller required the largest computation. Only the Strong-ARM and the Pentium III processors were faster than the CSP; they took 0.72 and 0.24 times respectively.

	4th order	7th order	13th order	46th order
CSP	1	1	1	1
TMS320C31	3.48	4.14	4.07	3.83
TMS320C54	9.10	7.6	6.59	5.97
C167	56.34	62.1	76.16	61.7
Strong-ARM	0.72	0.93	0.77	0.75
Pentium III	0.24	0.28	0.303	0.29

Table 7.9 Normalised computation time (CSP = 1)

To understand why the CSP is able to compete against some high performance processors and in fact to outperform some DSPs, we need to analyse closely the assembly code for each processor. The following paragraphs analyse segments of assembly code for some of the processors included in the benchmark.

Figure 7.6 shows one line of the C program that implements the 4th order filter and the corresponding assembly code for the TMS320C31 DSP.

```
// X[3] = X[3] + A[3]*X[2] + B[3]*U;
    LDFU         @0a03fh, R1
    LDFU         @0a049h, R0
    MPYF         @0a017h, R0
    MPYF         @0a01ah, R1
    ADDF         @0a04ah, R0
    ADDF3        R0, R1, R0
    STF          R0, @0a04ah
```

Figure 7.6 Segment of assembly code for the TMS320C31 DSP

A total of 7 instructions are needed to perform the operation. The first two instructions load data into the register file. Then, two multiply and one addition instructions are executed with operands read both from the memory and register file. A final addition of two values stored in registers produces the final result, which is then stored again in memory. As can be seen in the assembly code, this DSP can perform operations where some operands are read directly form memory. This reduces the number of load instructions that move data from memory to the register file, and as a consequence reduces the computation time.

Figure 7.7 shows the assembly code required to perform the same operation using the C167 microcontroller. Unlike the TMS320C31 DSP, the C167 requires the operands used for multiplications and additions to be stored in registers. Also the C167 can only handle 16-bit words. As a consequence, the code includes a large number of 'move' instructions to load/store memory data to/from the registers. The number of instructions is also increased by calling a subroutine that performs the multiplication and also because two instructions are needed per addition. A total 34 instructions are required to complete the operation, which combined with the number high average clock cycles per instruction and slow clock frequency, result in large computation times.

```
// X[3] = X[3] + A[3]*X[2] + B[3]*U;

     MOV        R6, DPP2:0x000C
     MOV        R7, DPP2:0x000E
     MOV        R4, DPP1:0x0034
     MOV        R5, DPP1:0x0036
     CALLA      CC_UC, ?C_LMUL(0x21E)
     MOV        R8, R4
     MOV        R9, R5
     ADD        R8, DPP2:0x0010
     ADDC       R9, DPP2:0x0012
     MOV        R4, DPP2:0x0024
     MOV        R5, DPP2:0x0026
     MOV        R6, R14
     MOV        R7, R15
     CALLA      CC_UC, ?C_LMUL(0x21E)
     ADD        R4, R8
     ADDC       R5, R9
     MOV        DPP2:0x0010, R4
     MOV        DPP2:0x0012, R5

?C_LMUL:
     MULU       R5, R6
     MOV        R5, DPP3:0x3E0E
     MULU       R7, R4
     ADD        R5, DPP3:0x3E0E
     MULU       R4, R6
     ADD        R5, DPP3:0x3E0C
     MOV        R4, DPP3:0x3E0E
     RET
```

Figure 7.7 Segment of assembly code for the C167 microcontroller

Figure 7.8 shows the assembly code required to perform the operation using now the Strong-ARM processor. This processor also required the operands to be stored in memory but it can handle 32-bit words. Thus, only a few load instructions are required. To complete the operation, only two multiply-accumulate instructions are required. The result is stored back into memory by a single store instruction. 8 instructions are required to complete the operation, which is one more than the number required by the TMS320C31. However, because of the high clock frequency at which this processor operates, the computation time is reduced significantly.

```
// X[3] = X[3] + A[3]*X[2] + B[3]*U;

      ldr   %r3,[%r10, #A+12-.LOOSTRING2]
      ldr   %ip,[%r9, #X+8-.LOOBSS]
      ldr   %r2,[%r9, #X+12-.LOOBSS]
      mla   %r2,%ip,%r3,%r2
      ldr   %r3,[%r10, #B+12-.LOOSTRING2]
      ldr   %r4,[%r8, #U-.LOODATA]
      mla   %r2,%r3,%r4,%r2
      str   %r2,[%r9, #X+12-.LOOBSS]
```

Figure 7.8 Segment of assembly code for the Strong-ARM processor

In contrast to the processors shown so far, the CSP is able to perform the operation with just two instructions. This is because all the operands are stored in the register file and therefore can be accessed without delay, and because the MAC instruction that completes the multiply-and-accumulate operation in a single cycle (see Figure 7.9).

```
// X[3] = X[3] + A[3]*X[2] + B[3]*U;

//CSP code                        Operation
MAC    X3, A3, X2, X3             // X[3] = A[3]*[X]2 + X[3]
MAC    X3, B3, U , X3             // X[3] = B[3]*U   + X[3]
```

Figure 7.9 CSP instructions example

To complement the benchmark, Table 7.10 shows how the CSP compares with the other processors in terms of complexity and power consumption. Technology and voltages supplies are also shown.

Processor	Technology	Complexity	I/O Power Supply	Core Power Supply	Power Consumption
CSP	0.25	12k gates	3.3 V	3.3 V	0.82 W
TMS320C31	0.6	5 M transistors	3.3 V	1.8 V	2.6 W
TMS320C54	0.6	*	5 V	5 V	*
C167	0.5	1.6 M transistors	5 V	5 V	1.5 W
Strong-ARM	0.35	525k gates.	3.3 V	2.0 V	1 W
Pentium III	0.25	28 M transistors	N/A	2.0 V	>20 W

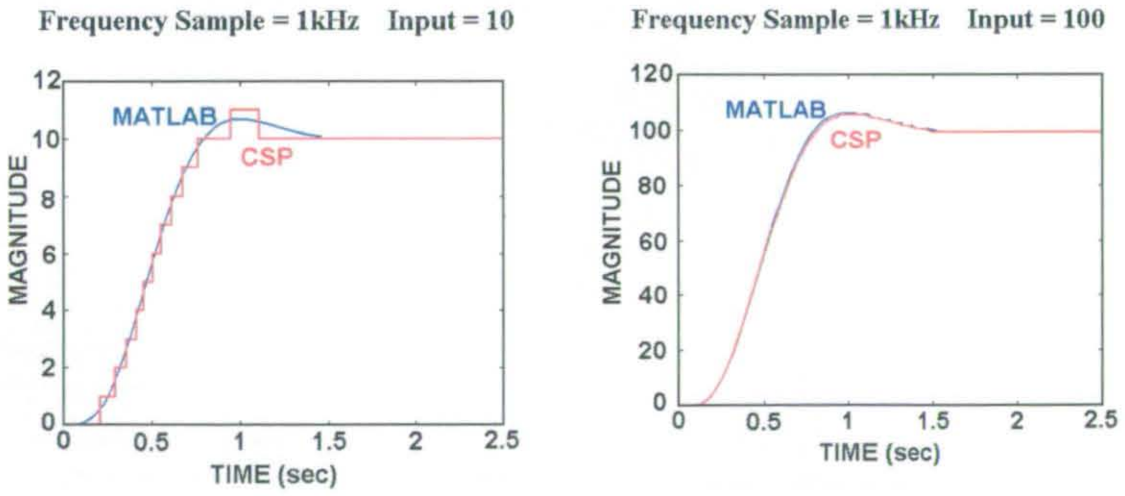
Table 7.10 Complexity and power consumption comparison

The power consumption is directly proportional to the clock frequency; thus it is possible to reduce the power consumption by reducing the clock frequency. Instead of clocking the CSP at the maximum possible speed, it is sufficient to use a clock frequency that will allow the CSP to perform the operations sufficiently fast to satisfy the requirements of the control system to be implemented.

7.6 SIMULATION RESULTS

This section shows some simulation results. The controller examples were implemented in the CSP and simulated using a variety of input signals and sample frequencies. The outputs of the CSP are compared with the 'ideal' results obtained with a Matlab program that uses standard full precision 32-bit floating-point format to represent the variables.

Figure 7.10 shows responses to step inputs of magnitude 10 and 100 sampled at 1kHz. These results indicate that the CSP's performance accuracy is very good. There is an inevitable quantisation effect with the smaller input, but it can be seen that the output is essentially following the ideal response.


Frequency Sample = 1kHz Input = 10

Frequency Sample = 1kHz Input = 100

Figure 7.10 Response of 4th order filter to step inputs of 10 and 100

Similar results were observed when sinusoid inputs were used. Figures 7.11 and 7.12 show responses to sinusoid inputs of 0.1 and 1 Hz. The maximum magnitude of the signals is 512 and the sample frequency is 1kHz.

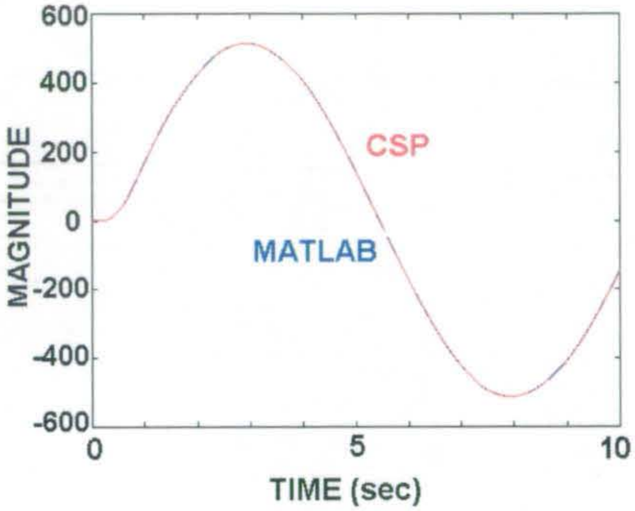


Figure 7.11 Response of 4th order filter to sinusoid input of 0.1Hz

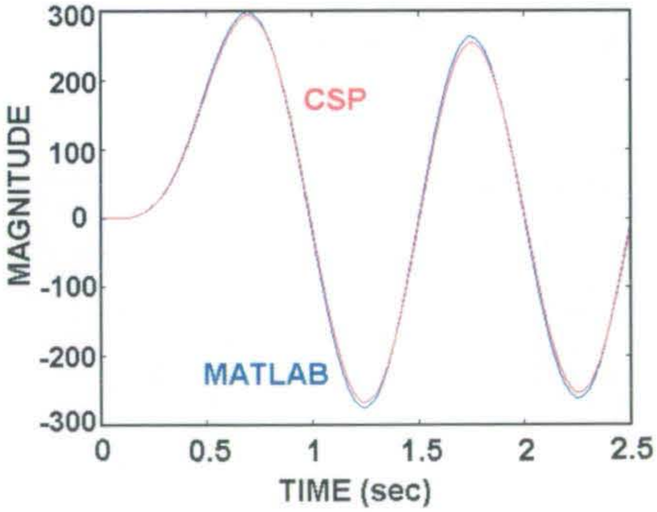


Figure 7.12 Response of 4th order filter to sinusoid input of 1Hz

As described in Chapter 4, the internal variable wordlength was chosen with a sufficient number of fractional bits to ensure that a response would be obtained with the smallest possible input (i.e. value of unity), over a very wide range of sample frequencies. But of course there is a limitation even with the most optimised numerical scheme. The differences observed in the outputs are the consequence of quantisation of both the coefficients and the variables within the CSP.

Figure 7.13 shows a set of step responses with small inputs at very high sample frequency, and graph (a) shows that with an input of 1 with a sample frequency of 10kHz no output is obtained. If the input is doubled at the same frequency, graph (b), some movement is seen on the output, although it is substantially different from the exact output. Alternatively, keeping the smallest input of 1 and having the sample frequency of 5kHz gives an output which, although coarsely quantised, is clearly following the general trend, see graph (c). The final graph (d) has an input of 2 with 5kHz, and is beginning to show a reasonable response.

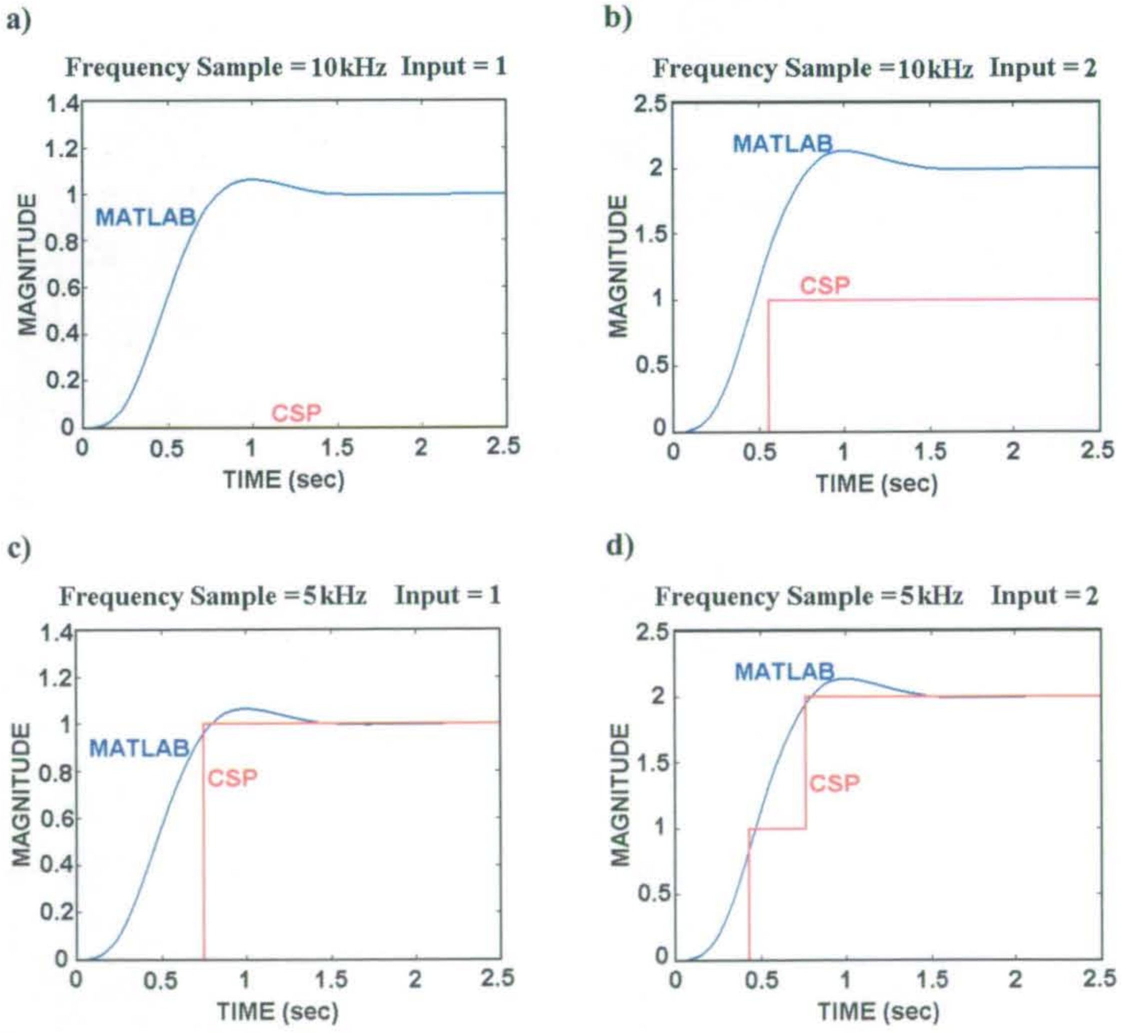


Figure 7.13 Response of 4th filter to step inputs of 1 and 2 sampled at 5 and 10kHz

It is important to realise these limiting conditions nevertheless represent an impressive performance - a fourth-order filter sampled at 10,000 times its cut-off frequency is extremely demanding for digital filtering applications, and is substantially more demanding than normally required for real-time control. A few more fractional bits in the variable wordlength would of course restore proper operation in the unlikely circumstance of it being necessary. For this particular example, 5 bits are needed to restore proper operation as can be seen in Figure 7.14. The results shown were obtained using 32-bit state variables (5 additional bits for underflow).

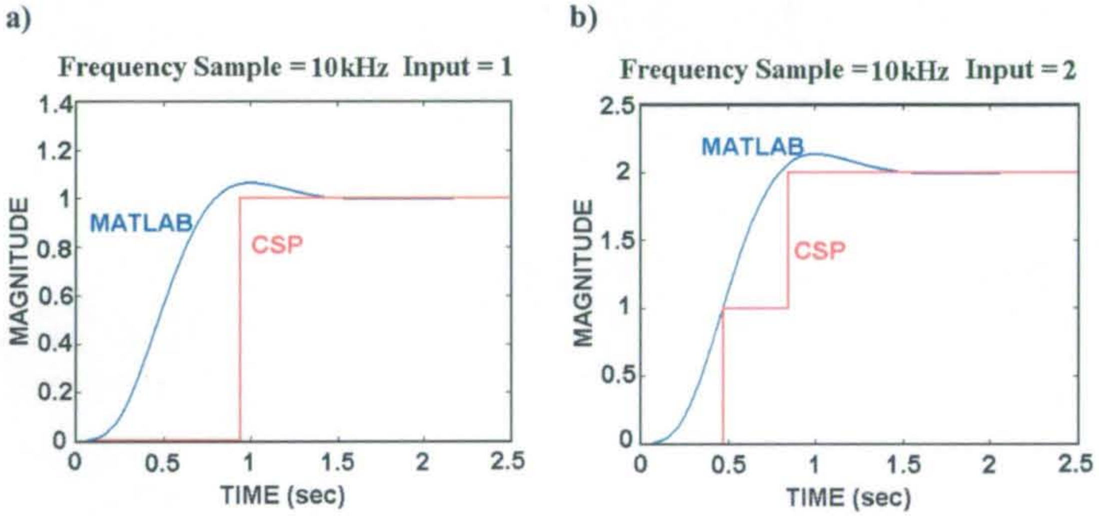


Figure 7.14 Response of 4th filter to step inputs of 1 and 2 sampled at 10kHz with 5 extra bits for underflow

The CSP was also simulated using the higher order controllers. It would be impractical to present the results of all simulations

Figure 7.15 shows the output response when step signals with magnitude of 512 are applied in parallel to the three inputs (see Figure 7.3) and the sample frequency is 1kHz. Figure 7.16 shows the responses when a sinusoid of amplitude of 512 and frequencies of 0.1 (7.12a) and 1Hz (7.12b) is applied to input 1. Step signals of magnitude 512 are applied to inputs 2 and 3, the sample frequency remains at 1kHz.

Finally, Figure 7.17 shows a response of the 46th order controller when step signals of magnitude 512 are applied to all the inputs. The sample frequency is 1kHz. Despite the complexity of these examples, the comparison of the CSP's output with the exact response remains excellent.

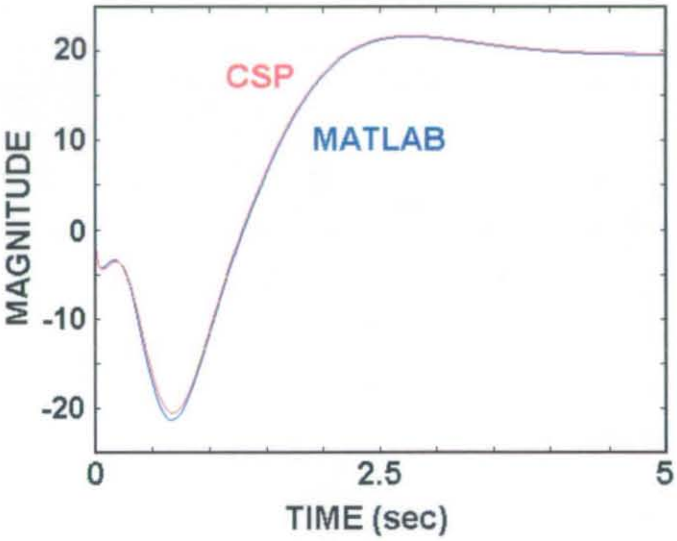


Figure 7.15 Response of the 13th order controller to step inputs of 512 sampled at 1kHz applied simultaneously to the three inputs

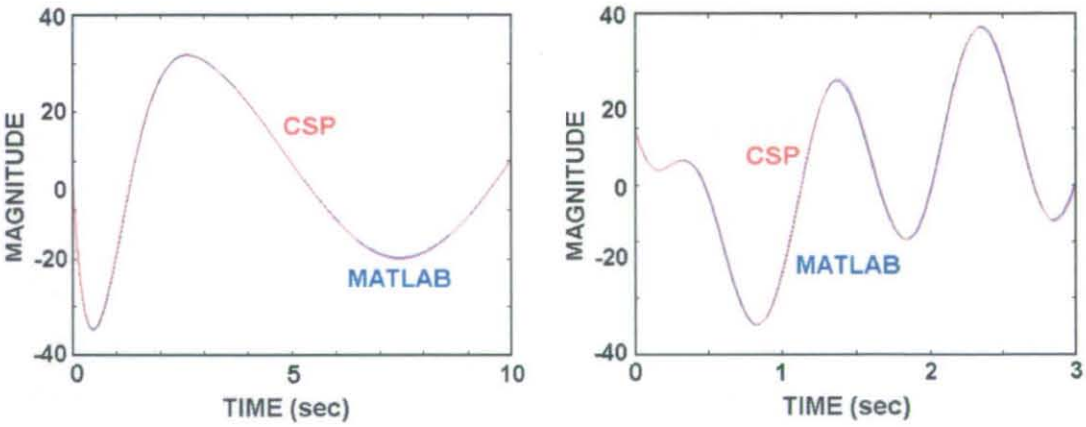


Figure 7.16 Response of the 13th order controller to a sinusoid input of 0.1 and 1 Hz sampled at 1kHz applied to input 1 and step inputs of magnitude 512 applied to inputs 2 and 3

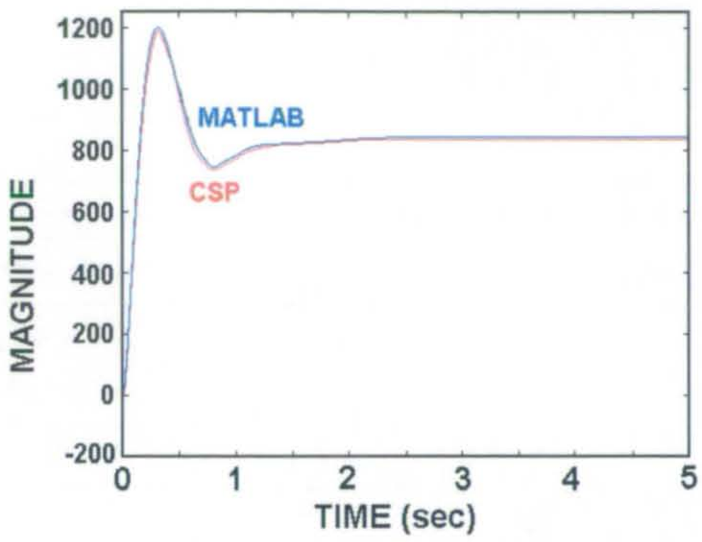


Figure 7.17 Output 1 response of the 46th order controller to step inputs of 512 sampled at 1kHz applied simultaneously to all three inputs

7.7 SUMMARY OF THE CHAPTER

The main features of the selected processors and controller examples used for the benchmark were described. The benchmark shows that the CSP can satisfy the requirements of a range of high sample rate controller examples. This is because of the good numerical properties of the algorithm combined with an architecture optimised to implement that algorithm.

The benchmark showed that the CSP outperforms some commercially available high-speed DSPs in terms of computation time and power consumption. An analysis of the assembly code revealed that fixed-point processors required up to 10 times more instructions than the CSP to implement the same algorithms. Even the floating-point processors required at least twice the number of instructions.

Also, simulation results where the CSP output is compared against the output of a Matlab program that uses full precision to represent the coefficients and internal variables were shown. The simulation showed that the CSP produces almost identical results to those obtained with the Matlab program.

As mentioned in Section 4.5.1, the coefficient format adopted to implement the CSP allows representing any coefficient with an accuracy of 1%, which is more that enough for most control applications [Forsythe91, Goodall92]. Thus, if the controller does not produce the expected results, additional bits should be added to the state variable format to restore proper operation.

Chapter 8

Conclusions

8.1 OBJECTIVES OF THE CHAPTER

This chapter concludes this thesis and evaluates the results obtained. The objectives of this chapter are:

- To review the original objectives of this thesis
- To present a summary of results and conclusion presented in previous chapters
- To discuss strengths and shortcomings of this work
- To present a framework of potential future work

8.2 REVIEW OF OBJECTIVES AND INVESTIGATIONS

The objective of this work is to investigate whether by providing customised hardware support for delta law control it is possible to provide a low-cost high-performance embedded controller.

From the review of existing approaches to implement digital control presented in Chapter 2 and from the analysis presented in Chapter 3, the following investigations were identified.

- The identification of efficient controller formulation to implement the control algorithm, which may be exploited to reduce the number of required operations and is suitable for hardware implementation

- The design of a hardware architecture to support the control algorithm and a software structure to implement the CSP program

- The identification a comprehensive set of tests to prove the CSP's operation for a variety of filters types over a range of input conditions.

8.3 Conclusions

An introduction to this thesis was presented in Chapter 1. It briefly discussed the problems faced by the control engineers when implementing high performance control systems using general-purpose processors. Also, it explained the potential benefits of using special-purpose architectures to implement such systems.

Chapter 2 defined digital control systems and presented a review of relevant background needed to appreciate this work. It presented a detailed analysis of the current approaches used to implement digital controllers and a review of relevant past work on special-purpose architectures specially those applied to control systems. The main conclusion of this chapter is that the best solution to implement a real-time high performance control systems largely depends on the particular requirements of each application. Factors like cost, performance, integration, easy of development, power consumption, development tools, will determine which option is the most suitable to implement a specific control system.

Chapter 3 describes the methodology and design flow used to implement the CSP. In Chapter 4, the controller formulation used to implement the CSP was introduced. It was shown that state-space approach using the δ operator offers a number of advantages when implementing control systems if compared with the traditional approaches. The reduced dynamic range of controller states and low coefficient sensitivity characteristics of this formulation results in a short internal variable and coefficient wordlength. Also, other proprieties of the algorithm that facilitate its hardware implementation where discussed.

Chapter 5 looks into the hardware implementation of the CSP. The design methodology involved the identification of the number and types of processing elements, the size and number of the memories, and the required communications channels. This general process starts with the specification of the main components and an overall system specification was defined as the design progressed. The proposed architecture takes advantage of the analysis of the control algorithm in Chapter 4 to permit efficient and cost effective realisations of the required processing functions.

Chapter 6 looks into the software that implements the control algorithms within the CSP and the software environment needed to support this implementation. It explains the software scheme adopted to implement the control algorithm and the CSP instruction set. The reduced number of operations required to implement the control loop results in a short CSP program, where the actual number of instructions is determined by the control system characteristics.

Finally, Chapter 7 presented the results of benchmarking the CSP against other processors running the same algorithms and the results of some simulations. It includes an explanation of the selected example controllers and processors used. It is shown that the CSP can satisfy a range of high sample rate controller examples due to its good numerical properties. The results showed that the CSP outperforms some commercially available high-speed DSPs by a significant margin. This is possibly due to a simplified hardware realisation that fully exploits the characteristics of the control algorithm.

8.4 Analysis of results

8.4.1 Achievements of this work

From the conclusions of each chapter we can conclude that the objectives of this work have been fulfilled. By identifying the requirements to implement real-time LTI control and by exploiting the numerical properties of the δ operator, a new method and processor architecture to implement real-time LTI controller has been proposed. The numerical properties of the controller formulation results in a stable, low instruction count algorithm.

The design of a simplified hardware multiply-and-accumulate unit results in a high-speed, low power, low cost numerically stable processor for embedded control. A comprehensive set of tests has shown that the CSP operates correctly on a variety of filter types over a range of input conditions. The results of a benchmark indicate that the control system processor outperforms some comercially available high-speed DSPs when implementing the example controllers. The control system processor was successfully implemented and verified on a programmable device.

The CSP is a compact, high-speed special purpose processor, which enables a low-cost solution to a wide range of LTI control problems. It offers a very effective implementation for embedded control and it is applicable to any solution of IIR filters. It is important to appreciate that, although the CSP outperforms some commercially available high-speed DSPs by a significant margin, it is much simpler. The modest gate count confers a number of advantages, namely reduced cost due to small die size and simpler packaging, and low power.

8.4.2 Limitation of this work

This section describes some limitations of this work.

- The proposed method and architecture only applies to the linear time-invariant controllers
- Although the CSP was successfully implemented into a ProASIC device and its functionality and speed verified, it was not tested in real control environments due to time limitations.
- Extra effort may be required to solve data dependency problems when programming the CSP

8.5 FUTURE WORK

This section presents some areas that have been identified as potential extension to this work. Also, some other possible approaches to implement a CSP processor are introduced.

8.5.1 Extension of current research

The CSP's dedicated architecture and careful numerical formulation ensure that it will perform deterministically in a real-time embedded control environment, although it is recognised that other functions are necessary in such applications for which the CSP is not well suited. It is necessary to address ways in which the variety of functions required for high-performance real-time control can be most effectively achieved.

The introduction of support for adaptive control would provide a more powerful and flexible alternative to implement real-time control. To achieve this, an adaptation mechanism that identifies certain characteristic parameters of the system has to be defined. Based on those parameters, the values of the coefficients can be modified to adjust the signal processing in order to minimise a previously adopted error measure at the output of the system.

The CSP can be considered as a specialised peripheral of a larger system included to relieve the general-purpose processor of performing fixed repetitive functions that can be performed more efficiently by dedicated hardware. It can also be integrated directly as an extra processing component within the general-purpose processor architecture. Furthermore, opportunities for the CSP are as an IP core to enable systems integrators to utilise its capabilities to provide high-performance computation as part of a more complex system-on-a-chip solution.

8.5.2 Other investigations

The implementation of the CSP concept can also be explored using the following approaches:

Single bit processing

Single bit processing is based on the use of bit-serial arithmetic and represents a viable alternative to the traditional bit-parallel arithmetic. A major advantage of using bit-serial arithmetic is that it significantly reduces chip area by eliminating wide buses and by using small processing elements. Additionally, Two's complement representation is suitable for use with bit-serial arithmetic.

Single instruction processor

The CSP can be seen as a single-instruction processor, where MAC is the only instruction. To achieve this, memory space can be partitioned into several sections where memory cells are associated with each input/output port. In this case, the processor only task is to move data between the MAC unit and memory cells. As there is only one instruction, no instruction decoding is necessary. Thus, the 'instruction' will only contain the source and destination address. This idea can be extended to include more that one processing element. In this case, special attention must be given to the schedule of operations to avoid data dependency problems and to use the available hardware resources efficiently.

Reconfigurable architectures

Reconfigurable architectures offer potential solutions to satisfy the demands of complex systems where a number the different functions required. They are based on two basic ideas. First, the architecture fits the algorithm and not vice versa, and second, to provide hardware support only for the algorithmic functions that are active at any particular time. This requires a device that can be configured 'on-the-fly' at very high speed. This concept, where algorithms are directly mapped onto dynamic hardware, is also known as adaptable computing.

8.6 SUMMARY

This chapter has presented the conclusions and main results of the investigations described in this thesis. A review of the objectives together with a summary of each chapter also been included. It presented a summary of the achievements of this work based on the objectives set on Chapter 1 and identified the limitations of this work.

Finally, this chapter identified future potential investigations to extend the work of this research and explained other possible approaches that can be used to implement the CSP concept.

Appendix A

General CSP program

A.1 GENERAL CSP PROGRAM

This appendix shows a general form of a CSP program. The actual number of instruction will depend on the order of the controller and the number of inputs and outputs. Note that the order in which the instructions within the control loop are executed may need to be rearrangement to avoid data dependency problems.

```
// Coefficient initialisation

    READ $C_0$, $IPt_0$, $DRC_0$            // Store constant 0 in coefficient format
    READ $C_1$, $IPt_0$, $DRC_1$            // Store constant 1 in coefficient format
    READ $C_{-1}$, $IPt_0$, $DRC_{-1}$      // Store constant -1 in coefficient format
    READ $a_1$, $IPt_0$, $DRa_1$            // Store matrix A
         :
    READ $a_n$, $IPt_0$, $DRa_n$
    READ $b_{1,1}$, $IPt_0$, $DRb_{1,1}$    // Store matrix B
         :
    READ $b_{n,\alpha}$, $IPt_0$, $DRb_{n,\alpha}$
    READ $c_{1,1}$, $IPt_0$, $DRc_{1,1}$    // Store matrix C
         :
    READ $c_{\beta,n}$, $IPt_0$, $DRc_{\beta,n}$
    READ $d_{1,1}$, $IPt_0$, $DRd_{1,1}$    // Store matrix D
         :
    READ $d_{\beta,\alpha}$, $IPt_0$, $DRd_{\beta,\alpha}$

// State variables and partial product initialisation

    READ $S_0$, $IPt_0$, $DRS_0$            // Store constant 0 in State Variable format
    READ $S_1$, $IPt_0$, $DRS_1$            // Store constant 1 in State Variable format

    READ $u_1$, $IPt_0$, $DRS_0$            // Initialise Input 1, $U_1 = 0$
       :                                        :
```

```
    READ $u_\alpha$, $IPt_0$, $DRS_0$            // Initialise Input 1, $U_\alpha = 0$

    READ $y_1$, $IPt_0$, $DRS_0$            // Initialise Output 1, $Y_1 = 0$
      :                                          :
    READ $y_\beta$, $IPt_0$, $DRS_0$            // Initialise Output 1, $Y_\beta = 0$

    READ $x_0$, $IPt_0$, $DRS_0$            // Initialise State variable 1, $X_1 = 0$
      :                                          :
    READ $x_n$, $IPt_0$, $DRS_0$            // Initialise State variable n, $X_n = 0$

    READ $p_1$, $IPt_0$, $DRS_0$            // Partial product 1, $P_1 = 0$
      :                                          :
    READ $p_\beta$, $IPt_0$, $DRS_0$            // Partial product $\beta$, $P_\beta = 0$

    READ s, $IPt_0$, $DRS_0$              // $\sigma_0 = 0$

    WRITEPC $PC_0$, rt                 // Initialise PC Start
    WRITEPC $PC_1$, rt                 // Initialise PC Stop

: BEGIN ALGORITHM CYCLE             // Algorithm cycle

// Read Sampled Inputs

    READ $u_1$, $IPt_1$                   // Copy Sample Input 1 to register file
      :                                          :
    READ $u_\alpha$, $IPt_\alpha$                   // Copy Sample Input $\alpha$ to register file

// Calculate $\mathbf{DU}_N$ and add the results to $\mathbf{CX}_N$ (stored as partial product) to obtain the output
// data $\mathbf{Y}_N$

    MAC $p_1$, $d_1$, $u_1$, $p_1$              // Execute the first product of $\mathbf{DU}_N$ needed to
                                        // calculate $Y_1$ and add to the partial product
                                        // obtained in the previous algorithm cycle.
    MAC $p_1$, $d_{12}$, $u_2$, $p_1$             // Execute the rest of products to obtain $Y_1$
      :
    MAC $y_1$, $d_{1\alpha}$, $u_\alpha$, $p_1$             // Last product to calculate $Y_1$

    // Repeat the previous instructions to calculate $Y_2, \ldots, Y_\beta$
    // At this step, the output data vector $\mathbf{Y}_N$ is available.

// Copy Outputs values to Output Ports

    WRITE $OPt_0$, $y_1$                 // Copy Output 1
      :
    WRITE $OPt_{\beta-1}$, $y_\beta$               // Copy Output $\beta$

// Calculate $\mathbf{X}_{n+1} = \mathbf{AX}_N + \mathbf{BU}_N$

    // Calculate $X_{n,N+1}$
    MAC $x_{n,N+1}$, $a_n$, $x_{n-1,N}$, $x_{n,N}$      // Execute the first MAC operation to calculate
                                        // $X_{n,N+1}$
    MAC $x_{n,N+1}$, $b_{n,1}$, $u_1$, $x_{n,N+1}$     // Execute the second MAC operation to calculate
    MAC $x_{n,N+1}$, $b_{n,2}$, $u_2$, $x_{n,N+1}$     // $X_{n,N+1}$
```

```
        :
    MAC $x_{n,N+1}$, $b_{n,\alpha}$, $u_\alpha$, $x_{n,N+1}$            // Last product to calculate $X_{n,N+1}$
    MAC acc, $C_0$, $S_0$, $x_{n,N+1}$                  // First addition to calculate $\sigma_{N+1}$

    // Calculate $X_{n-1,N+1}$
    MAC $x_{n-1,N+1}$, $a_{n-1}$, $x_{n-2,N}$, $x_{n-1,N}$      // Execute the first MAC operation to calculate
                                                // $X_{n-1,N+1}$
    MAC $x_{n-1,N+1}$, $b_{n-1,1}$, $u_1$, $x_{n-1,N+1}$      // Execute the second MAC operation to calculate

    MAC $x_{n-1,N+1}$, $b_{n-1,2}$, $u_2$, $x_{n-1,N+1}$      // $X_{n-1,N+1}$
        :
    MAC $x_{n-1,N+1}$, $b_{n-1,\alpha}$, $u_\alpha$, $x_{n-1,N+1}$      // Last product to calculate $X_{n-1,N+1}$
    MAC acc, $C_1$, acc, $x_{n-1,N+1}$                // Second addition to calculate $\sigma_{N+1}$

    // Repeat the previous instructions to calculate $X_{n-3,N+1}, \ldots, X_{2,N+1}$

    // Calculate $X_{1,N+1}$
    MAC $x_{1,N+1}$, $a_1$, s, $x_{1,N}$                  // Execute the first MAC operation to calculate
                                                // $X_{1,N+1}$
                                                // (use $\sigma_N$)
    MAC $x_{1,N+1}$, $b_{11}$, $u_1$, $x_{1,N+1}$              // Execute the second MAC operation to calculate
                                                // $X_{1,N+1}$
    MAC $x_{1,N+1}$, $b_{12}$, $u_2$, $x_{1,N+1}$
        :
    MAC $x_{1,N+1}$, $b_{1\alpha}$, $u_\alpha$, $x_{1,N+1}$              // Last product to calculate $X_{1,N+1}$

    // When the $X_{1,N+1}$ value is obtained, the MAC instruction will copy the result of acc1+
    // $X_{1,N+1}$ directly to the register that contains $\sigma_{N+1}$

    MAC s, $C_1$, acc, $x_{1,N+1}$                    // Last addition to calculate $\sigma_{N+1}$

// Calculate $CX_{N+1}$ and store as partial product

    MAC $p_1$, $c_{11}$, $x_{1,N+1}$, $S_0$                  // Execute the first MAC operation to calculate
                                                // partial product 1 ($P_1$)
    MAC $p_1$, $c_{12}$, $x_{2,N+1}$, $p_1$
        :
    MAC $p_1$, $c_{1n}$, $x_{n,N+1}$, $p_1$                  // Execute the last MAC operation to calculate
                                                // partial product 1 ($P_1$)

    // Repeat the previous instructions to calculate $P_2, \ldots, P_\beta$

: END ALGORITHM CYCLE
```

Where:

u_i is the address of input data U_i
y_i is the address of output data Y_i
$x_{i,N}$ is the address of state variable $X_{i,N}$
p_i is the address of partial product P_i
a_i is the address of coefficient A_i
$b_{i,j}$ is the address of coefficient $B_{i,j}$
$c_{i,j}$ is the address of coefficient $C_{i,j}$

$d_{i,j}$ is the address of coefficient $D_{i,j}$
s is the address of σ value
acc is the address of accumulator value
$X_{i,N}$ is the value of state variable i at time N
$\mathbf{U}_N$ is the input vector at time N
$\mathbf{X}_N$ is the state variable vector at time N
$\mathbf{Y}_N$ is the output vector at time N
IPt_i is the input channel i
OPt_i is the output channel i
DRx is the address of variable x in memory Data ROM
C_k is the constant k stored in Coefficient format
S_k is the constant k stored in State Variable format

Appendix B

Sets of Coefficients used for simulations

This appendix shows the set of coefficients for the controllers used to obtain the results shown in Section 7.6.

B.1 4TH ORDER 1HZ BUTTERWORTH LOW PASS FILTER

Sample frequency = 100Hz

$$\mathbf{A}=\begin{bmatrix} 1 & 2.1446e-2 & 0 & 0 \\ 0 & 1 & 4.3459e-2 & 0 \\ 0 & 0 & 1 & 8.6931e-2 \\ -1.7591e-1 & -1.7591e-1 & -1.7591e-1 & 8.2409e-1 \end{bmatrix} \quad \mathbf{B}=\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1.7591e-1 \end{bmatrix}$$

$$\mathbf{C}=\begin{bmatrix} 3.9676e-5 & 1.3991e-3 & 4.2951e-2 & 1 \end{bmatrix} \qquad \mathbf{D}=\begin{bmatrix} 8.9206e-7 \end{bmatrix}$$

Sample frequency = 1kHz

$$\mathbf{A}=\begin{bmatrix} 1 & 2.2340e-3 & 0 & 0 \\ 0 & 1 & 4.4330e-3 & 0 \\ 0 & 0 & 1 & 8.8665e-3 \\ -1.7594e-2 & -1.7594e-2 & -1.7594e-2 & 9.8241e-1 \end{bmatrix} \quad \mathbf{B}=\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1.7594e-2 \end{bmatrix}$$

$$\mathbf{C}=\begin{bmatrix}4.3807e-8 & 1.4855e-5 & 4.4679e-3 & 1\end{bmatrix} \qquad \mathbf{D}=\begin{bmatrix}9.6556e-11\end{bmatrix}$$

Sample frequency = 5kHz

$$\mathbf{A}=\begin{bmatrix}1 & 4.4852e-4 & 0 & 0\\ 0 & 1 & 8.8814e-4 & 0\\ 0 & 0 & 1 & 1.7764e-3\\ -3.5186e-3 & -3.5186e-3 & -3.5186e-3 & 9.9648e\text{-}1\end{bmatrix} \quad \mathbf{B}=\begin{bmatrix}0\\ 0\\ 0\\ 3.5186e-3\end{bmatrix}$$

$$\mathbf{C}=\begin{bmatrix}3.5357e-10 & 5.9736e-7 & 8.9679e-4 & 0.9997\end{bmatrix} \qquad \mathbf{D}=\begin{bmatrix}1.5558e-13\end{bmatrix}$$

Sample frequency = 10kHz

$$\mathbf{A}=\begin{bmatrix}1 & 2.2389e-4 & 0 & 0\\ 0 & 1 & 4.4417e-4 & 0\\ 0 & 0 & 1 & 8.8842e-4\\ -1.7593e-3 & -1.7593e-3 & -1.7593e-3 & 9.9824e-1\end{bmatrix} \quad \mathbf{B}=\begin{bmatrix}0\\ 0\\ 0\\ 1.7593e-3\end{bmatrix}$$

$$\mathbf{C}=\begin{bmatrix}4.3659e-11 & 1.4973e-7 & 4.5033e-4 & 1.0071\end{bmatrix} \qquad \mathbf{D}=\begin{bmatrix}9.7700e-15\end{bmatrix}$$

B.2 7TH ORDER TWO-INPUT TWO-OUTPUT CONTROLLER

Subsystem 1

$$\mathbf{A}=\begin{bmatrix}1 & 1.42857e-1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 1.0507e-3 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 2.3407e-3 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 3.6662e-3 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 5.8833e-3 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 1.01006e-2\\ -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & 9.765e-1\end{bmatrix}$$

$$\mathbf{B}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\2.349e-2\end{bmatrix}\quad \mathbf{C}=\begin{bmatrix}-23.9359 & -5.22161 & 9.9371 & 8.32588 & 6.40424 & 4.371 & 2.12578\end{bmatrix}$$

$$\mathbf{D}=\begin{bmatrix}-13.06409\end{bmatrix}$$

Subsystem 2

$$\mathbf{A}=\begin{bmatrix}1 & 1.42857e-1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 1.0507e-3 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 2.3407e-3 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 3.6662e-3 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 5.8833e-3 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 1.01006e-2\\ -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & 9.765e-1\end{bmatrix}$$

$$\mathbf{B}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\2.349e-2\end{bmatrix}\quad \mathbf{C}=\begin{bmatrix}-16.172 & -13.529 & -5.721 & -4.791 & -3.678 & -2.4803 & -1.142\end{bmatrix}$$

$$\mathbf{D}=\begin{bmatrix}6.6721\end{bmatrix}$$

Subsystem 3

$$\mathbf{A}=\begin{bmatrix}1 & 1.42857e-1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 1.0507e-3 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 2.3407e-3 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 3.6662e-3 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 5.8833e-3 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 1.01006e-2\\ -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & 9.765e-1\end{bmatrix}$$

$$\mathbf{B}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\2.349e-2\end{bmatrix}\qquad \mathbf{C}=\begin{bmatrix}1.7864 & 0.7149 & 4.1203 & 3.621 & 2.997 & 2.2303 & 1.288\end{bmatrix}$$

$$\mathbf{D}=\begin{bmatrix}-5.2864\end{bmatrix}$$

Subsystem 4

$$\mathbf{A}=\begin{bmatrix}1 & 1.42857e-1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 1.0507e-3 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 2.3407e-3 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 3.6662e-3 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 5.8833e-3 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 1.01006e-2\\ -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & -2.349e-2 & 9.765e-1\end{bmatrix}$$

$$\mathbf{B}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\2.349e-2\end{bmatrix}\qquad \mathbf{C}=\begin{bmatrix}14.824 & 5.253 & 1.8937 & 1.7257 & 1.5807 & 1.3668 & 1.0128\end{bmatrix}$$

$$\mathbf{D}=\begin{bmatrix}-3.82474\end{bmatrix}$$

B. 3 13TH ORDER THREE-INPUT ONE-OUTPUT MAGLEV LOOP CONTROLLER

System parameters

wi = 1	Accelerometer integrator frequency (rad/s)
wib = 2	Flux integrator frequency (rad/s)
ws = 8	Suspension filter frequency (rad/s)
G = 10	Main loop gain
k = 5	Main loop phase advance ratio
taw = 0.01	Main loop phase advance time constant (s)

wn = 120	Main loop notch filter (rad/s)
tawb = 0.01	Flux loop PI time constant (s)
tawh = 0.001	Flux loop high freq. filter time constant (s)
Sample frequency = 1kHz	

Subsystem 1

$$\mathbf{A} = \begin{bmatrix} 1 & 0.004 & 0 \\ 0 & 1 & 0.008 \\ -0.016 & -0.016 & 0.984 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 0.016 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 0.0079 & 1 & 1 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 3.1809e-5 \end{bmatrix}$$

Subsystem 2

$$\mathbf{A} = \begin{bmatrix} 0.90625 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0.09375 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} -0.375 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 0.48437 \end{bmatrix}$$

Subsystem 3

$$\mathbf{A} = \begin{bmatrix} 1 & 0.1071 \\ -0.1264 & 0.8736 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0.1264 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} -0.8364 & 0.0564 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 0.9436 \end{bmatrix}$$

Subsystem 4

$$\mathbf{A} = \begin{bmatrix} 1 & 5.0e-4 & 0 \\ 0 & 1 & 0.001 \\ -0.002 & -0.002 & 0.998 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 0.002 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 5.0e-4 & 0.4996 & -1.0e-4 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 2.4975e-7 \end{bmatrix}$$

Subsystem 5

$$\mathbf{A} = \begin{bmatrix} 1 & 0.0014 \\ -0.0028 & 0.9972 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0.0028 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} -0.3561 & -5.0e-4 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 4.993e-4 \end{bmatrix}$$

Subsystem 6

$$\mathbf{A} = \begin{bmatrix} 1 & 0.00 \\ -0.6667 & 0.3333 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0.6667 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 0 & -6.0048 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 0.35 \end{bmatrix}$$

B.4 46TH ORDER TWELVE-INPUT FOUR-OUTPUT MAGLEV VEHICLE CONTROLLER

System parameters

wi = 1	Accelerometer integrator frequency (rad/s)
wib = 2	Flux integrator frequency (rad/s)
wsb = 8	Bounce suspension filter frequency (rad/s)
wsp = 6	Pitch suspension filter frequency (rad/s)
wsr = 5	Roll suspension filter frequency (rad/s)
Gb = 10	Bounce loop gain
kb = 5	Bounce loop phase advance ratio
tb = 0.01	Bounce loop phase advance time constant (s)
wnb = 120	Bounce loop notch filter (rad/s)
Gp = 10	Pitch loop gain
kp = 5	Pitch loop phase advance ratio
tp = 0.01	Pitch loop phase advance time constant (s)
wnp = 120	Pitch loop notch filter (rad/s)
Gr = 10	Roll loop gain
kr = 5	Roll loop phase advance ratio
tr = 0.01	Roll loop phase advance time constant (s)
wnr = 120	Roll loop notch filter (rad/s)
tawb = 0.01	Flux loop PI time constant (s)
tawh = 0.001	Flux loop high Freq. filter time constant (s)
Sample frequency = 1kHz	

Subsystems 1, 2, and 3

$$\mathbf{A} = \begin{bmatrix} 1 & 3.3976e-3 & 0 \\ 0 & 1 & 7.9839e-3 \\ -1.5999e-2 & -1.5999e-2 & 0.984001 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 1.5999-2 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 7.9283e-3 & 0.9999 & 0.999 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 3.1809e-5 \end{bmatrix}$$

Subsystem 4, 5 and 6

$$\mathbf{A} = \begin{bmatrix} 0.904762 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 9.5238e-2 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} -3.8095 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 4.8095 \end{bmatrix}$$

Subsystems 7, 8 and 9

$$\mathbf{A} = \begin{bmatrix} 1 & 5.4545e-1 \\ -2.6148e-2 & 0.973851 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 \\ 2.6148e-2 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} -4.486e-1 & -5.9429e-3 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 9.9405e-1 \end{bmatrix}$$

Subsystems 10, 11, 12 and 13

$$\mathbf{A} = \begin{bmatrix} 1 & 4.9962e-4 & 0 \\ 0 & 1 & 79.9974e-4 \\ -1.9999e-3 & -1.9999e-3 & 0.998 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 1.9999-3 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 3.4992e-3 & 4.9962e-1 & -2.4975e-7 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 2.4975e-5 \end{bmatrix}$$

Subsystems 14, 15, 16 and 17

$$\mathbf{A} = \begin{bmatrix} 1 & 1.4265e-3 \\ -2.8e-3 & 0.9971 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 2.8e-3 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 3.5613e-1 & -4.993e-4 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 4.993e-4 \end{bmatrix}$$

Subsystems 18, 19, 20 and 21

$$\mathbf{A} = \begin{bmatrix} 1 & -1.6653e-1 \\ -6.6666e-1 & 0.3333 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 6.6666e-1 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 7.5-e1 & -6.00479 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 3.4999e-1 \end{bmatrix}$$

References

[Ackenhusen99] John G. Ackenhusen, "Real-time signal processing: design and implementation of signal processing systems", pp. 25-76, Prentice Hall, 1999.

[Actel00a] Actel ProASIC A500K Family User's Guide. Actel Corp. 2000.

[Actel00b] MEMORYmaster User's Guide. Actel Corp. 2000.

[Agrawal95] J. P. Agrawal, E. Bouktache, O. Farook and C. R. Sekhar, "Hardware software system design of a generic embedded controller for industrial applications," Conference record of the 1995 IEEE Industry applications conference, Vol. 3, pp. 1887-1892, 1995.

[Baugh73] C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," IEEE Transactions of Computers, C-22, pp. 1045-1047, Dec. 1973.

[Bdti00] "Choosing a DSP processor", white paper, Berkeley Design Technology, Inc., www.bdti.com.

[Cady97] Frederick M. Cady, "Microcontrollers and microcomputers, principles of software and harware engineering", pp. 4-24, Oxford university press, 1997.

[Catthoor91] F. Catthoor, F. Franssen, K, Cools, C. Hendriks, F. Demeester, J. De Schutter and H. De Man, "An application-specific microcoded architecture for a robot control application", VLSI Signal Processing, IV, pp. 452-461, IEEE Pres, 1991.

[Chen91] D. C. Chen and J. Rabaey, "PADDI: Programmable Arithmetic Devices for DIgital Signal Processing", VLSI Signal Processing, IV, pp. 240-249, IEEE Pres, 1991.

[Costa97] A. Costa, A. De Gloria, F. Giudici and M. Olivieri, "Fuzzy logic microcontroller," IEEE Micro, Vol. 17, Issue 1, pp. 66-74, Jan.-Feb. 1997.

[Darbyshire95] E. P. Darbyshire and C. J. Kerry, "A multiprocessor architecture for large scale real-time control", IEE Colloquium on Multiprocessor DSP (Digital Signal Processing) - Applications, Algorithms and Architectures. 1995.

[Dettlof89] Wayne D. Dettloff and Hiroyuki Watanabe "A Fuzzy Logic Controller with Reconfigurable, Cascadable Architecture", IEEE 1989.

[Donald94] Donald L. Hung "Custom design of a hardware fuzzy logic controller", IEEE World Congress on Computational Intelligence. Proceedings of the Third IEEE Conference on Fuzzy Systems, 1994.

[Eyre98] Jennifer Eyre, Jeff Bier, "DSP processors hit the mainstream", IEEE computer, August 1998.

[Eyre00] Jennifer Eyre, Jeff Bier, "The evolution of DSP processors", IEEE signal processing magazine, March 2000.

[Feuer96] Arie Feuer, Graham C. Goodwin, "Sampling in digital signal processing and control", pp. 122-245, Birkhäuser, 1996.

[Forsythe91] W. Forsythe and R. M. Goodall, "Digital control: Fundamentals, theory and practice," pp. 122-170, McGraw-Hill, 1991.

[Fujioka96] Y. Fujioka, M. Kameyarna, N. Tomabechi, "Reconfigurable parallel VLSI processor for dynamic control of intelligent robots" IEE Proc.-Comput. Digit. Tech. Vol. 143, No. 1, January 1996.

[Furber99] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. day, J. Liu and N. Paver, "AMULET2e, an asynchronous embedded controller," Proceedings of the IEEE, Vol. 87, No. 2, February 1999.

[Garberg96] B. Garbergs and B. Sohlberg, "Specialised hardware for state space control of a dynamic process," Proceedings of the 1996 IEEE TENCON- Digital Signal Processing Applications, Vol. 2, pp. 895-899, 1996.

[Garberg98] B. Garbergs and B. Sohlberg, "Implementation of a state space controller in FPGA", 9th Mediterranean Electrotechnical Conference, MELECON 98, 1998.

[Goodwin92] Graham C. Goodwin, Richard H. Middleton, H. Vincent Poor, "High-speed digital signal processing and control", Proceedings of the IEEE, Vol. 80, No. 2, February 1992.

[Goodwin01] Graham C. Goodwin, Stefan F. Graebe, Mario E. Salgado, "Control system design", pp. 1-33, Prentice Hall, 2001.

[Goodall78] R M. Goodall, Williams R A and Barwick R W, “Ride quality specification and suspension controller design for a magnetically suspended vehicle,” Proceedings InstMC Symp on Dynamic Analysis of Vehicle Ride and Maneuvering Characteristics, pp 79-89, Nov 1978.

[Goodall85] R. M. Goodall and D. S. Brown, "High speed digital controllers using an 8-bit microprocessor," Software and Microsystems, vol. 4, pp. 109-116, 1985.

[Goodall90] R. M. Goodall, "The delay operator Z-1 - inappropriate for use in recursive digital filters?", Transactions of the Institute of Measurement and Control, Vol 12, No5, 1990.

[Goodall92} R. M. Goodall, "A practical method for determining coefficient word length in digital filters", IEEE Transactions on signal processing, Vol. 40, No. 2, April 1992.

[Goodall93] R. M. Goodall and B. J. Donoghue, "Very high sample rate digital filters using the δ operator" IEE Proceedings-G, 140, pp. 199-206, 1993.

[Goodall00] R. M. Goodall, "Perspectives on processing for real-time control," Proceedings of IFAC workshop AARTC2000, Palma de Mallorca, Spain, May 2000.

[Grout95] I. A. Grout, S. E. Burge and A. P. Dorey, "Design and testing of a PI controller ASIC", Microprocessors and Microsystems, Vol. 19, No.1, pp. 15-22, Feb 1995.

[Herpel93] H.-J. Herpel, N. Wehn, M. Gasteier and M. Glesner, "A reconfigurable computer for embedded control applications," Proceedings of the IEEE workshop on FPGAs for Custom Computing Machines, pp. 111-120, 1993.

[Infineon00] "C167 Derivatives, User's manual", Infineon Technologies AG, www.infineon.com.

[Intel00a] "Pentium III processor Data sheet", Intel Corp. www.intel.com.

[Intel00b] "StrongARM-110 Microprocessor Data sheet", Intel Corp. www.intel.com.

[Irwin98] George W. Irwin, "Computing & control: back to the future", Computing & control engineering journal, IEE, February 1998.

[Jaswa85] V.C. Jaswa, C. E. Thomas and J. Pedicone, "CPAC: Concurrent processor architecture for control," IEEE Transactions on computers, vol. 34, pp. 163-169, 1985.

[Lapsley97] Phil Lapsley, Jeff Bier, Amit Shoham, Edward A. Lee, "DSP processor fundamentals", IEEE Press, 1997.

[Lang84] J. H. Lang, "On the design of a special-purpose digital control processor", IEEE Transactions on Automatic Control, Vol. AC-29, No.2, March 1984.

[Ling88] Y. L. C. Ling, P. Sadayappan, "A VLSI robotics vector processor for real-time control" Proceedings of the 1988 IEEE International Conference on Robotics and Automation, 1988.

[Liu91] J. Liu, Z. Q. Mao, G. Z. Lu and W. H. Han, "A new VLSI architecture for real-time control of robot manipulators," Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Vol. 2, pp. 1828-1835, April 1991.

[Liu99] J. Liu, M. Brooke, "A fully parallel learning neural network chip for real-time control, " International Joint Conference on Neural Networks, IJCNN '99, Vol. 4, pp. 2323-2328, 1999.

[Martin98] Daniel Martin, Robert O. Owen, "A RISC architecture with uncompromised digital signal processing and microcontroller operation", Proceedings of the 1998 IEEE International Conference on Speech and Signal Processing, 1998.

[Middleton90] R. H. Middleton and G. C. Goodwin (1990), "Digital control and estimation – a unified approach," pp. 54-82, 456-481, Prentice Hall, 1990.

[Nadehara95] K. Nadehara, M. Hayashida and I. Kuroda, "A low-power, 32-bit RISC processor with signal processing capability and its multiply-adder", Workshop on VLSI Signal Processing, VIII, IEEE Signal Processing Society, 1995.

[Nekoogar99] Farzad Nekoogar, Gene Moriarty, "Digital control using digital signal processing", pp. 1-24, Prentice Hall, 1999.

[Nise00] Norman S. Nice, "Control systems engineering", pp. 1-33, 703-747, The Benjamin/Cummings Publishing Company, Inc., 2000.

[Palmer94] Richard P. Palmer, Peter A. Rounce "An architecture for application specific neural network processors", Computing & Control Engineering Journal, pp. 260-264, December 1994.

[Patyra96] Marek J. Patyra, Janos L. Grantner and Kirby Koster, "Digital Fuzzy Logic Controller: Design and implementation", IEEE Transactions on Fuzzy Systems, Vol. 4, No. 4, November 1996.

[Pirsch98] P. Pirsch, "Architectures for Digital Signal Processing," pp. 245-283, 305-353, Wiley, 1998.

[Predko99] Michael Predko, "Title Handbook of microcontrollers" McGraw-Hill,1999.

[Proakis96] John G. Proakis, Dimitris G. Manolakis, " Digital signal processing: principles, algorithms, and applications", Prentice Hall, 1996.

[Samet98] L. Samet, N. Masmoudi, M. W. Kharrat and L. Kamoun, "A digital PID controller for real-time and multi loop control: a comparative study," 1998 IEEE International conference on Electronics, Circuits and Systems, Vol. 1, pp. 291-296, 1998.

[Santina94] Mohammed S. Santina, Allen R. Stubberud, Gene H. Hostetter, "Digital control system design", pp. 490-566, Saunders College Publishing, 1994.

[Schlett98] Manfred Schlett, "Trends in embedded-microprocessor design", IEEE computer, August 1998.

[Spray91] A. Spray and S. Jones, "PACE: A regular array for implementing regularly and irregularly structured algorithms," IEE Proceedings-G, vol. 138, pp. 613-619, 1991.

[Texas00a] "TMS320C3x User's guide", Texas Instruments Inc., www.ti.com

[Texas00b] "TMS320C54x User's guide", Texas Instruments Inc., www.ti.com

[Tokhi95] M. O. Tokhi and M. A. Hossain, "Parallel DSP for real-time control", IEE Colloquium on Multiprocessor DSP (Digital Signal Processing) - Applications, Algorithms and Architectures. 1995.

[Tsunekawa95] Yoshitaka Tsunekawa, Mamoru Miura, "High-performance VLSI architecture suitable for control systems for state-space digital filters using distrubuted arithmetic", Electronics and communications in Japan, Part 3, Vol. 78, No. 5, 1995.

[Wanhammar99] Lars Wanhammar, "DSP integrated circuits", pp. 1-27, 225-267, Academic Press, 1999.

Publications

R. Goodall, S. Jones, R.A. Cumplido-Parra, F. Mitchell, S. Bateman, "A Control System Processor Architecture For Complex LTI Controllers", Proceedings, 6th IFAC Workshop on Algorithms and Architectures for Real-Time Control, (AARTC 2000), Palma de Mallorca, Spain, May 2000.

Rene A. Cumplido-Parra, Simon R. Jones, Roger M. Goodall, Fiona Mitchell and Stephen Bateman,"High Performance Control System Processor", Proceedings of the 3rd Workshop on System Design Automation - SDA 2000, Dresden, Germany, March 2000.

Previous Paper selected for publication on: "System Design Automation: Fundamentals, Principles, Methods, Examples", Edited by Renate Merker and Wolfgang Schwarz, Kluwer Academic Publishers, ISBN 0-7923-7313-8, pp. 140-151, March, 2001.

Roger Goodall, Simon Jones and Rene Cumplido-Parra, "Digital Filtering for High Performance Real-Time Control," IEE Colloquium on Digital Filters: An enabling technology, London, April 1998.

René Cumplido, Simon Jones, Roger Goodall and Stephen Bateman "A High Performance Processor for embedded Real-Time Control" Submitted to IEEE Transactions on Control System Technology.