High Performance Control System Processor

René A. Cumplido-Parra[1], Simon R. Jones[1], Roger M. Goodall[1] and Stephen Bateman[2]
[1]Loughborough University, UK; [2]Gatefield Corporation, USA

Abstract

This paper describes a compact, high-speed special purpose processor, which offers a low-cost solution to implement linear time invariant controllers. The controller has been reformulated into a modified state-space representation based on the δ operator, which is optimised for numerical efficiency. This Control System Processor (CSP) has been implemented using a programmable ASIC (ProASIC) device.

1 Introduction

There is now a variety of control design methods by which appropriate control laws can be created for complex multi-variable systems, but the actual implementation of control laws is a part of the design process which most control engineers want to achieve as straightforwardly and transparently as possible. One approach is to programme a fixed point microprocessor (μP) device in a high level language, using floating point variables so that numerical issues are not a concern, but the computational overhead is large and surprising restrictions in sample rate are found. A second approach is to use a fixed point digital signal processor (DSP) which has an architecture better targeted for computationally-intensive applications, and these offer some speed advantage over a μP. Other options are to use a number of parallel processors or sophisticated floating-point DSPs, but this doesn't result in a cost-effective solution, especially for high-volume embedded control applications.

The difficulty is that there are particular numerical requirements in control system processing for which standard processor devices are not well suited, in particular arising from the high sample rates which are need to avoid adverse effects of sample delays upon stability. These could be satisfied in either μP or DSP devices using "hand-crafted" numerical routines, probably written in assembler language, but as mentioned above control engineers generally have neither the will nor the skill to do this. 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.

Continuing improvements in microelectronic technology has made feasible large reprogrammable silicon chips (Field Programmable Gate Arrays – FPGAs) which can be configured to realise complex computational systems without incurring either the delay or the costs associated with custom silicon. As a result electronic designers and control engineers are looking once again at the potential of designing low-cost, high-performance special-purpose hardware for embedded real-time control. With current chip complexities of up to 2 million designable gates and circuit density growth of 100% every 18 months it appears that the complexity of algorithm is limited only by design capability and not by silicon complexity.

1.1 The use of targeted architectures

It is well accepted that customisation of silicon offers cost and performance advantages over standard components. Furthermore FPGAs open up this market to a much smaller product volume. Contemporary microprocessors offer high-performance at the price of increased cost and power consumption. Furthermore, there is good evidence that the extensive use of floating-point numbers in calculation results in large chips and slow operation. Our view is that be taking a considered view of the numerical and calculation requirements of the algorithm allows special purpose processors to be considered which provide well-targeted support of control laws. Such systems are likely to be smaller, cheaper, faster and lower power than conventional signal processors. In the past there has been much work on special purpose processors for control (e.g. Jaswa's CPAC [1], PACE [2]) but while they are intriguing ideas the cost of producing custom silicon proved prohibitive for initial exploitation and restricted experimentation with different architectural constructs. With High-level design tools such as VHDL and logic synthesis CAD suites allied to large low-cost reprogrammable FPGAs, the constraints no longer apply and we can now develop this area with full enthusiasm.

The remainder of this paper is structured as follows. Section 2 reviews the state-space representations of a controller, describes the formulation used to implement the CSP and the associated numerical issues. Section 3 describes the CSP architecture and its core. Section 4 details the CSP instruction set, program structure and software suite. Section 5 presents some simulation results, and finally section 6 concludes.

2 Control Issues

2.1 State-Space equations

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. 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.

With the arrival of modern applications, requirements for control systems increased in scope. Modelling systems by using linear, time-invariant differential equations and subsequent transfer functions became inadequate. The state-space approach is a unified method for modelling, analysing, and designing a wide range of systems.

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 and a second set of mathematical equations relating the state variables and the current input to the output signal.

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. 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. The general form of the state-space equations, to which all forms of control systems can be converted, is:

$$\mathbf{X}_{N+1} = \mathbf{A}\mathbf{X}_N + \mathbf{B}\mathbf{U}_N \qquad (1)$$

$$\mathbf{Y}_N = \mathbf{C}\mathbf{X}_N + \mathbf{D}\mathbf{U}_N \qquad (2)$$

2.2 Formulation used for the CSP

To implement the control algorithm we decided to adopt a modified structure based on the delta operator which present a number of advantages; among them, it does not require the long coefficient word-lengths needed to cope with high coefficient sensitivity associated with the z-operator.

The numerical problems associated with discrete-time control, in which the sample frequency will typically be two orders of magnitude higher than the controller bandwidth, are well known when the z operator is used [3]. This problem arises specially when sampling at high speed is needed, mostly because the difference between current and next input and output values may be increasingly small. Similarly it is recognised that the use of the δ operator overcomes a number of these problems, in which case the state equation becomes

$$\delta X = A_\delta X + B_\delta U \qquad (3)$$

This is sometimes defined as $\delta = (z-1)T$ (where T is the sample period), in which case there is a unification between discrete and continuous time since $\delta \to s$ (the Laplace operator) as $T \to 0$ [4]. In fact for the relatively high sample frequencies found for practical controllers $\delta = s$ is quite a realistic approximation and the coefficients in A_δ and B_δ become almost independent of the sample period. The effect of sample period must of course be taken into account when implementing δ, and an alternative simpler definition is to use $\delta = z - 1$ [5], in which case the correspondence between δ and s is lost but implementation is more direct

The modified canonic δ form affects the representation of the A and B matrices used to calculate the next value of the state variables. A large number of zeroes present in the A matrix can reduce the overall computation time, this is achieved by expanding and rearranging the matrices to reduce the number of operation required. The general form of the actual control equations, which will be implemented, is

$$\begin{aligned} X(n+1)T &= AX(nT) + BU(nT) \\ Y(nT) &= CX(nT) + DU(nT) \end{aligned} \qquad (4)$$

The modified canonic δ form is illustrated diagrammatically in figure 1 for a fourth-order SISO controller.

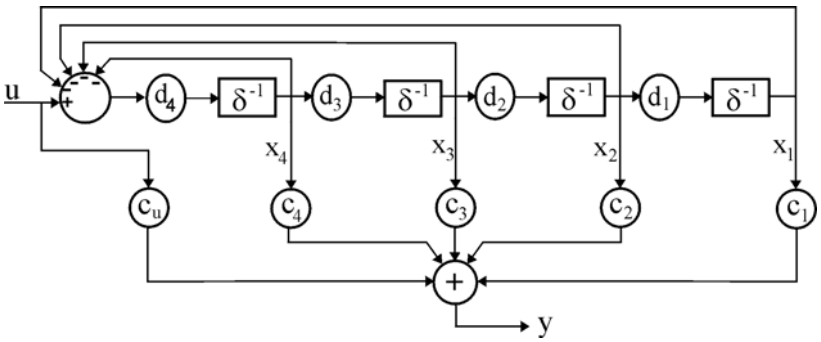

Figure 1. Modified canonic δ formulation

The corresponding state equations are:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_{n+1} = \begin{bmatrix} 1 & d_1 & 0 & 0 \\ 0 & 1 & d_2 & 0 \\ 0 & 0 & 1 & d_3 \\ 1-d_4 & -d_4 & -d_4 & -d_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_n + \begin{bmatrix} 0 \\ 0 \\ 0 \\ d_4 \end{bmatrix} u_n \qquad (5)$$

$$y_n = \begin{bmatrix} c_1 & c_2 & c_3 & c_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_4 \\ x_4 \end{bmatrix}_n + \begin{bmatrix} c_u \end{bmatrix} u_n$$

and the actual equations used for real-time implementation are as below; firstly the calculation of the output, then an update of the states ready for the next sample:

$$\begin{aligned} y &:= c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4 + c_u u \\ x_{temp} &:= x_1 + x_2 + x_3 + x_4 \\ x_1 &:= x_1 + d_1 x_2 \\ x_2 &:= x_2 + d_2 x_3 \\ x_3 &:= x_3 + d_3 x_4 \\ x_4 &:= x_4 - d_4 x_{temp} + d_4 u \end{aligned} \qquad (6)$$

Notice that x_{temp} is used to store the sum of the old values of x_1 to x_4, which thereby avoids having to retain old values for the states while the new values are calculated – the state variables are then simply overwritten at each calculation.

2.3 Numerical requirements

Using this standard controller formulation the requirements for coefficients and controller state variables become relatively standardised across a wide range of applications, and these are illustrated in figure 2.

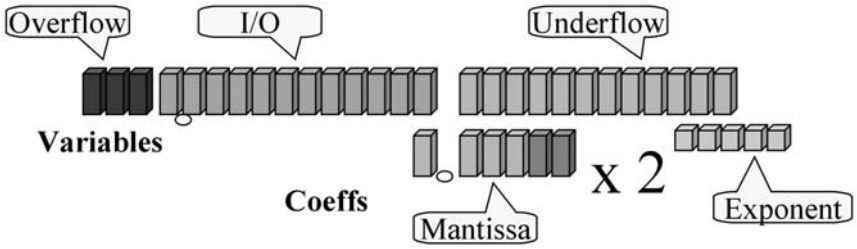

Figure 2. Numerical specification

The variables are 27 bit fixed point, with the input values brought in as integers, a small 3-bit allowance for overflow (although this is a nominal requirement because of the good scaling properties mentioned previously), and 12 fractional bits for underflow.

The coefficients are held in a simple low-precision floating-point form, with 6 bits for the mantissa and 5 bits for the exponent. In general the coefficients fractional with values which become progressively smaller as the sample frequency is increased, but a positive exponent is provided to implement greater than unity gains, a few of which are associated with most controllers.

This numerical specification will implement successfully the vast majority of LTI controller examples, and allows for the sample frequency being at least three orders of magnitude higher than the lowest pole in the controller. Of course if there are exceptional requirements it is always possible to reprogram the CSP hardware in the FPGA, maintaining the essential principles but extending the hardware precision as required. An example of implementing extremely high sample frequency digital filters using the modified canonic δ approach can be found in [6].

3 Hardware Architecture

3.1 CSP architecture

Figure 3 shows a block diagram for the CSP system. The core of the CSP comprises a simplified datapath with storage and computation capabilities. The Register bank stores all the constants, coefficients, state variables, inputs data, output data and partial products needed to perform the calculations.

The computation of the output values is done iteratively executing multiply-accumulation (MAC) operations. These operations are specified by the instructions fetched from an internal program ROM and decoded by the instruction handler. The instruction format contains the source and destination addresses of the operands used in the MAC operation. The program counter addresses the next instruction in program memory to be executed. An internal Data ROM contains the coefficients and the initial values for the state variables and program counter registers. A group of analogue-digital and digital-analogue converters provide the interface to the physical system being controlled. Figure 4 shows the processor interface, grey lines indicate data values while black lines indicate control signals.

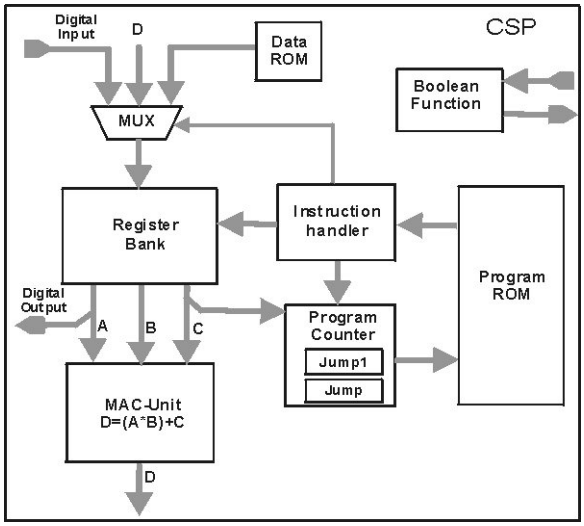

Figure 3. Processor architecture

The processor will be embedded within the complete control system and will normally be programmed in a separate programming system.

Important features of this architecture are:

- reduced precision of the variables when compared to full IEEE floating-point representation
- different numerical representations of coefficients and state variables which are satisfactory for a wide-range of controllers
- targeted MAC unit optimised to for calculating sum of products

This novel architecture combined with the use of a small and specialised instruction set presents cost and performance benefits for control applications over traditional architectures.

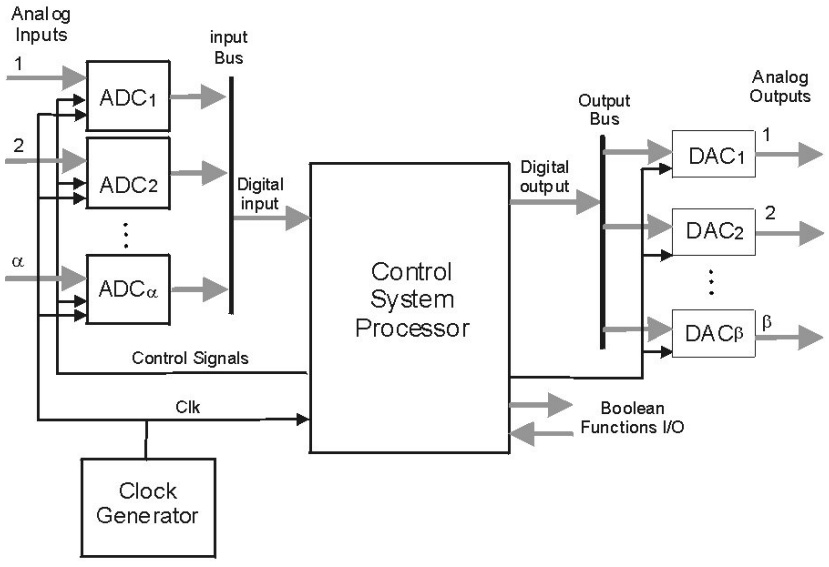

Figure 4. Processor interface

3.2 Core description

The core of the CSP includes a special-purpose multiply-accumulator unit (MAC) and a 4-port register bank (3 read, 1 write). The MAC unit executes the multiply-accumulate operations required to perform the control algorithm, i.e. D=A*B+C (see figure 5). The A input is in coefficient format (12 bits) and the B and C values are in variable format (27 bits).

A detailed low level design has been used to speed up the MAC operation. The system is pipelined such there is a latency of 4 clock cycles between instructions issues and the result being written back to the register bank. The compiler ensures that instruction dependencies are observed through an appropriate series of instruction issues.

The coefficient is split into its mantissa and exponent sections. The multiplier block multiply a state variable by the mantissa, the product is then shifted by a number of bits determined by the coefficient exponent. Finally, the result is added to other state variable to produce the output.

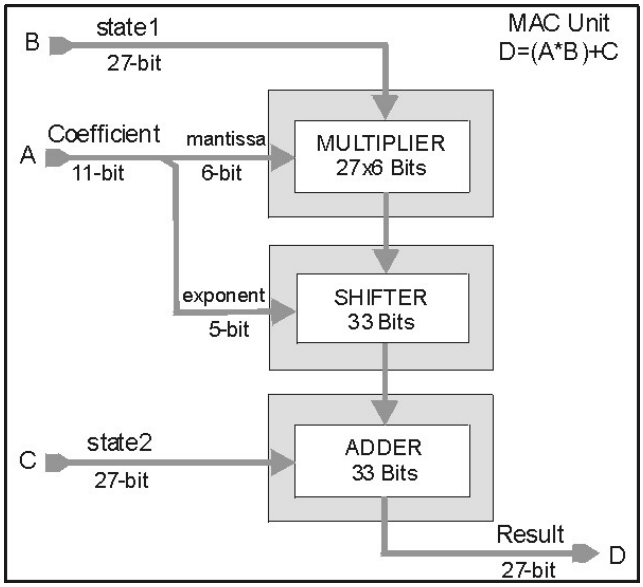

Figure 5. MAC Unit

3.3 Speed and Complexity

Table 1 shows the CSP complexity in terms of ProASIC tiles and equivalent gates. Everything except the program and data ROM are fixed in size; these are hardwired, and their size and speed depends upon the control algorithm being implemented. The figures shown are for the controller specified in section 2.2.

The synthesis of the CSP results in an overall gate count of fewer than 21,000 gates and a delay of 20ns, this allows a clock frequency of 50 MHz. The register bank is implemented using 9 embedded RAM blocks provided by ProASIC devices. Each block contains a 256 word deep by 9 bits wide memory, with 2 ports (1 read, 1 write). As such the CSP is a compact low cost core capable of implementing the most demanding real-time systems.

Block	Tiles (ProAsic)	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 1. CSP complexity

The maximum sampling frequency for a specific control system is determined by its complexity, i.e. the number of instructions needed to calculate the next state and output values.

The relatively small size of the processor core leaves much of the FPGA free such that it can be used to carry out other functions typically associated with real-time control – logical interlocking functions, background tasks such as gain-scheduling etc.

4 CSP Software

4.1 Instruction Set

Now we describe the operation of the CSP from the programmer's point of view. The CSP instruction set is very simple and specialised; it is targeted towards high-speed computation. Due to the MAC unit contains one pipe stage, multiple instructions can be overlapped in execution. At the time that the operands specified by one instruction are being read from the register bank and copied to the MAC unit inputs, the results obtained from the previous instruction is obtained at the output of the MAC unit and copied back to the register bank.

The processor only has four instructions (see table 2). The MAC instruction executes a multiply-accumulation operation on the operands indicated by the source addresses and stores the result in the destination address. This instruction allows performing the matrix multiplication accordingly to the state-space representation of the control system. Because the constants 0, 1 and -1 are stored in the register bank, other calculations can be mapped into this format. The MAC instruction can be used to add two values, increment a value by one, invert the sign of a value, simulate a no operation instruction, copy the contain of one regis7ter, etc. (e.g. D=C is achieved by setting A to 0).

There are no conditional jumps in the system. Unconditional jumps are supported for user programming. However, the code generator flattens all but the exterior loop. The program counter starts at zero increments until it reaches the value stored in the ‘jump1 register’, it is then reset to the value in the ‘jump2 register’. This permits an initial start up sequence to be followed and then a main loop to be repeatedly executed.

The READ instruction allows loading initial values from the data ROM into the register bank during the initialisation process. Also, when the algorithm loop has begun, this instruction is

used to read sampled input data from the input bus and to indicate the conversion time for the ADC's. Finally, the WRITE instruction is used to transfer the output values to the output bus.

The program is stored in the same chip as the processor together with simple Boolean functions for interlocking etc. A/D and D/A interfaces are also provided.

Instruction	Function	Description
MAC d, s1, s2, s3	(s1*s2) + s3 -> d	Multiply accumulation operation
WRITEPC sel, s	s -> pc[sel]	Write to program counter registers (PC,jump1,jump2)
READ d, in_pt, s	If in_pt =0 DataRom[s] -> d else Input[in_pt] -> d	Read from Data ROM or input ADC
WRITE out_pt, s	s -> output[out_pt]	Read from Register Bank and write to output DAC

Table 2. CSP instructions

4.2 Program Structure

To generate the CSP Program it is necessary consider the order in which operations must be done, number of inputs, number of outputs and order of the control system to be implemented. Although the CSP program is modified accordingly to the system to be controlled, the program scheme remains the same. It is divided in two main parts: initialisation and algorithm loop. In the initialisation part, all the coefficients and state variable initial values are transferred from the external data ROM to the register bank. 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. One of the main goals of this software scheme is to achieve a constant sample rate. The sub-tasks contained within the CSP program are listed below. Additionally, the numbers of CSP instructions required to perform each task are provided.

Task	Number of CSP instructions
Load Coefficients	$n + n\alpha + \beta n + \alpha\beta + 3$
Initialise State Variables and PC	$n + 2\beta + \alpha + 6$
: *Algorithm cycle start*	
Get Input Data	$\alpha + 2$
Calculate DU_N	$\alpha\beta$
Calculate Yn	β
Calculate X_N	$(2 + \alpha)n$
Calculate AX_N	
Calculate BU_N	
Calculate CX_{N+1}	$n\beta$
Supplementary Processes	$\beta + 1$
Write Output Data	
: *End Algorithm cycle*	

where α, β and n are defined as the number of input channels, the number of output channels and the number of internal state variables respectively.

4.3 Software suite

4.3.1 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. The CSP model architecture is modular; this modularity allows us to replace processing elements to perform 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. 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.

4.3.2 CSP signal generator

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. The CSP signal generator provides input test data to the CSP model. The signal streams are stored in binary files using double format. The type of signal, number of samples, magnitude, sample frequency, etc. are indicated by parameters sent to the program in the command line. The double format used to store data in the files is compatible with Matlab double formats, so these test files can be used as inputs in any part of the design process.

4.3.3 CSP program generator

The CSP program is created using the program generator. The sequence of instructions needed to perform the control algorithm is that illustrated in section 4.2. The number of instructions varies accordingly with the number of inputs, outputs and order of the system. Each calculation part is generated using these parameters to modify the source and destination addresses for each instruction. Also, the program generator rearrange the order in which the instructions are executed to avoid the data dependency problems associated with pipelined designs.

4.3.4 Programming the CSP

When the CSP program is correct, code can be generated in text format. This code is then converted into VHDL (as a ROM element) and added to the VHDL code. This is then synthesised and placed and routed. Note that this implies that system clock speed can be influenced by program complexity. The chip is then placed in the system board and operation commences on an asserted start signal. The processor is implemented in an Actel ProAsic FPGA (figure 6) which is a flash-programmable non-volatile device. Figure 7 shows the programmer and test system.

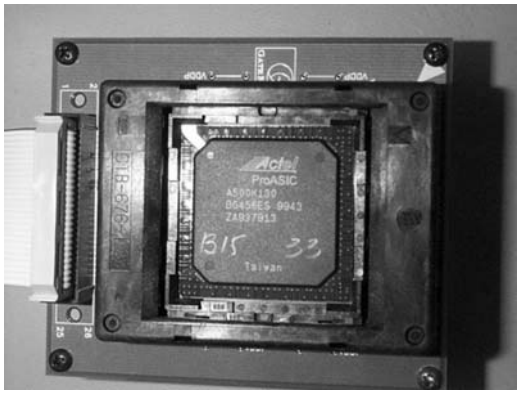

Figure 6. Actel ProASIC device

Figure 7. Programmer and test system

5 Results

Results of some simulations of the CSP are shown. These results have been cross-referenced against the results obtained with a Matlab program. Consider a fourth order 1Hz Butterworth filter as that of section 2.2 with a sample frequency of 100Hz. The coefficient values are:

$d_1 = 0.022,\ \ d_2 = 0.0446,\ \ d_3 = 0.088,\ \ d_4 = 0.178$

$c_1 = c_2 = c_3 = c_u = 0 \ \text{ and } \ c_4 = 1$

Two normalised input data stream were used in the simulations (i.e. VE_{min}= -1 and VE_{max}=1).

1. Step input $\quad u(t) = \begin{cases} 0.5 & t > 0 \\ 0 & \text{otherwise} \end{cases}$

2. Sin input $\quad u(t) = \begin{cases} 0.5 * \sin(t) & t > 0 \\ 0 & \text{otherwise} \end{cases}$

Figures 8 and 9 show the output and state variable values of the CSP VHDL Model and Matlab program with the step and sine signal as input respectively. Clearly, the values obtained with the two approaches are very similar. This demonstrates that the CSP specification is correct and can be implemented in practical applications.

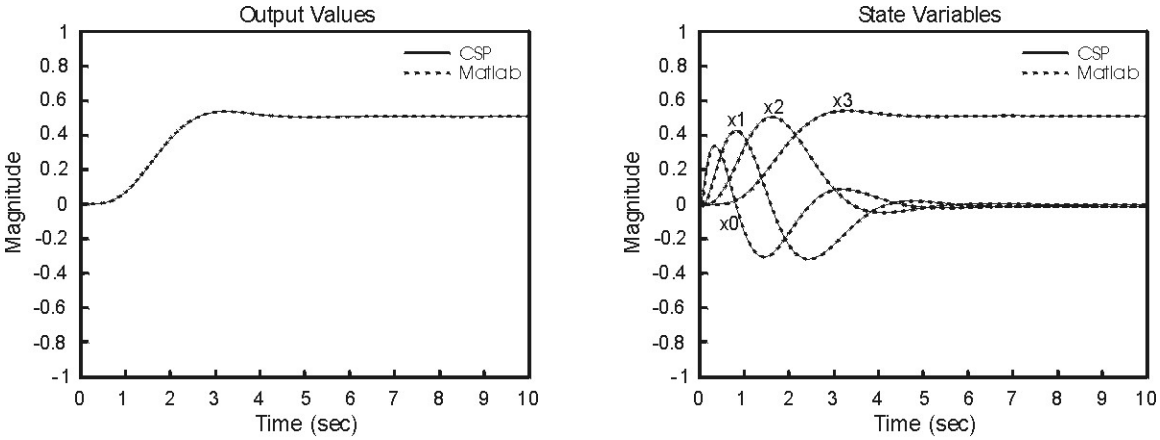

Figure 8. Output values and state variables for input signal 1

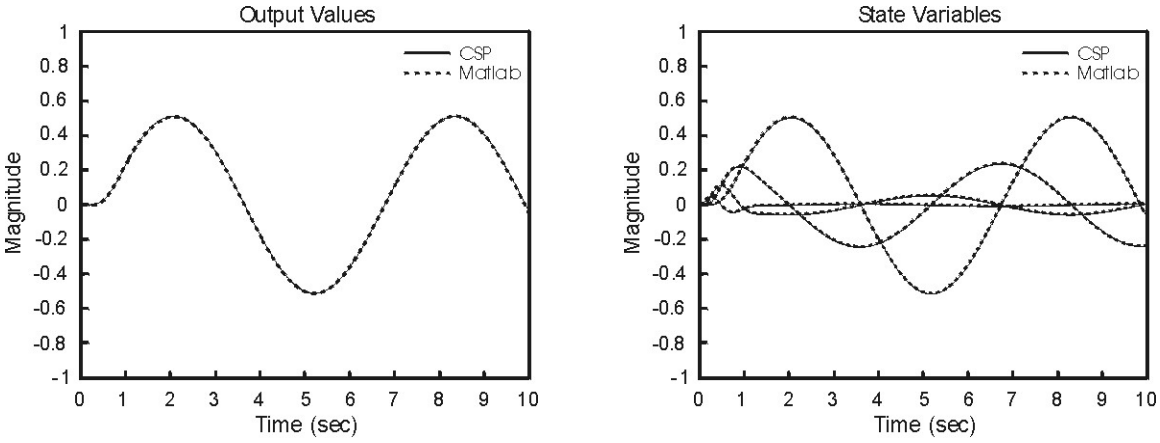

Figure 9. Output values and state variables for input signal 2

The time to perform a single MAC operation is one clock cycle, which is 20ns. The number of instructions needed to perform an algorithm cycle, which includes calculating the next state variables and outputs values is 23 (see section 4.2). Thus the total time required per algorithm cycle is 0.46 μs. This gives a maximum sampling frequency of 2.16MHz.

6 Conclusions

In this paper has been shown a special-purpose control system processor as a solution to implement complex linear time invariant controllers. The processor is implemented in an Actel ProAsic FPGA which is a flash-programmable device (non-volatile) together with the appropriate programmer, offering a one-chip solution. Further work is under way to construct a practical CSP demonstrator and test the design in a range of applications, including MAGLEV suspension control and aircraft flight control.

7 References

[1] Jaswa, V.C., Thomas, C.E., & Pedicone, J. (1985). CPAC: Concurrent processor architecture for control. IEEE Transactions on computers, C-34, 163-169.

[2] Spray, A., & Jones, S. (1991). PACE: A regular array for implementing regularly and irregularly structured algorithms. IEE Proceedings, Pt G, Vol. 138, No. 5, pp 613-619.

[3] Liu B, “Effect of finite wordlength on the accuracy of digital filters – a review”, IEEE Trans Circuit Theory, 1971, CT-18, (6), pp 670-677.

[4] Middleton R H and Goodwin G C, “Digital control and estimation – a unified approach” (Prentice Hall, 1990)

[5] Goodall R M and Brown D S, “High speed digital controllers using an 8-bit microprocessor”, Software and Microsystems, Vol. 4, Nos. 5 & 6, pp 109-116, Dec 1985.

[6] Goodall R M and Donaghue B, “Very high sample rate digital filters using the operator”, Proceedings IEE, Pt G, Vol. 140, No 3, pp 199-206, June 1993.